

Fall 1996

A Computational Paradigm on Network-Based Models of Computation

Venkatavasu Bokka
Old Dominion University

Follow this and additional works at: https://digitalcommons.odu.edu/computerscience_etds



Part of the [Programming Languages and Compilers Commons](#), and the [Theory and Algorithms Commons](#)

Recommended Citation

Bokka, Venkatavasu. "A Computational Paradigm on Network-Based Models of Computation" (1996). Doctor of Philosophy (PhD), dissertation, Computer Science, Old Dominion University, DOI: 10.25777/ktzv-ak04
https://digitalcommons.odu.edu/computerscience_etds/71

This Dissertation is brought to you for free and open access by the Computer Science at ODU Digital Commons. It has been accepted for inclusion in Computer Science Theses & Dissertations by an authorized administrator of ODU Digital Commons. For more information, please contact digitalcommons@odu.edu.

A COMPUTATIONAL PARADIGM ON NETWORK-BASED MODELS OF COMPUTATION


by


Venkatavasu Bokka
Indian Institute of Technology, Delhi, India


A Dissertation Submitted to the Faculty of
Old Dominion University in Partial Fulfillment of the
Requirements of the Degree of

DOCTOR OF PHILOSOPHY
COMPUTER SCIENCE
OLD DOMINION UNIVERSITY
November 1996

Approved by:


Stephan Olariu (Advisor)


James L. Schwing (Advisor)

 Larry Wilson

Alex Pothen


Przemysław Bogacki

ABSTRACT

A COMPUTATIONAL PARADIGM ON NETWORK-BASED MODELS OF COMPUTATION

Venkatavasu Bokka

Old Dominion University, 1996

Advisors: Drs. Stephan Olariu and James L. Schwing

The maturation of computer science has strengthened the need to consolidate isolated algorithms and techniques into general computational paradigms. The main goal of this dissertation is to provide a unifying framework which captures the essence of a number of problems in seemingly unrelated contexts in database design, pattern recognition, image processing, VLSI design, computer vision, and robot navigation. The main contribution of this work is to provide a computational paradigm which involves the unifying framework, referred to as the Multiple Query problem, along with a generic solution to the Multiple Query problem.

To demonstrate the applicability of the paradigm, a number of problems from different areas of computer science are solved by formulating them in this framework. Also, to show practical relevance, two fundamental problems were implemented in the C language using MPI. The code can be ported onto many commercially available parallel computers; in particular, the code was tested on an IBM-SP2 and on a network of workstations.

Copyright
by
Venkatavasu Bokka
1996

To my Parents

ACKNOWLEDGMENTS

This work could not be completed without the help of many individuals, to whom I would like to express my appreciation. First and foremost, I would like to thank my advisors, Drs. Stephan Olariu and James Schwing, who have put a great deal of time and effort into the guidance of this work. I would like to thank Dr. Schwing and Dr. Pothen for their help in conducting the experiments on IBM-SP2 at NASA Langley.

Next, I would like to convey my sincere thanks to the other members of my dissertation committee, Drs. Larry Wilson, Alex Pothen and Przemyslaw Bogacki. Their expertise, thorough reviewing and valuable suggestions have also led to a greatly improved dissertation.

I wish to extend my appreciation to the faculty of the department, and my fellow students for providing a stimulating research environment.

I would like to thank my brother Vijay Kumar and my friends Ramani G. N., Ranjita M., and Rao S. V., for their constant encouragement. I am grateful to my family for their support. I would like to especially mention Sri. P. Janaki Ramulu, my high school tutor, to whom I owe my academic achievements. Finally, special thanks to my sister, Usha.

TABLE OF CONTENTS

1	INTRODUCTION	1
1.1	Overview	1
1.2	State of the Art	7
1.3	Models of Computation	9
2	THE COMPUTATIONAL PARADIGM ON THE ACM	15
2.1	A Generic Multiple Query Algorithm	16
2.2	Rank-Related Computations	23
2.2.1	The Multiple Rank Problem	23
2.2.2	Histogram Computation	25
2.3	The Multiple Point Location Problem	27
2.4	Proximity-Related Computations	35
2.4.1	The Multiple Closest Segment Problem	35
2.4.2	The Multiple Circle Problem	37
2.4.3	The Multiple Range Problem	39
2.4.4	The Multiple Closest Point Problem	41
2.5	Stabbing-Related Problems	43
3	THE SORTED MATRIX ALGORITHM ON THE ACM	48
3.1	BSR Algorithm on the ACM	51

4	THE COMPUTATIONAL PARADIGM ON THE MMB	66
4.1	Lower Bounds	67
4.1.1	The Gather Problem	67
4.1.2	Lower bounds for instances of the MQ problem	68
4.2	A Generic Multiple Query Algorithm on MMB	76
4.3	Rank-Related Computations	81
4.3.1	The Multiple Rank Problem	81
4.3.2	Histogram Computation	85
4.4	The Multiple Point Location Problem	86
4.5	Proximity-Related Computations	89
4.5.1	The Multiple Closest Segment Problem	90
4.5.2	The Multiple Range Problem	90
4.5.3	The Multiple Circles Problem	91
4.5.4	The Multiple Closest Point Problem	91
4.6	Stabbing-Related Problems	92
5	THE SORTED MATRIX ALGORITHM ON THE MMB	96
5.1	Lower Bound	99
5.2	A Time-Optimal BSR Algorithm	102
6	IMPLEMENTATION AND CONCLUSIONS	118
6.1	Implementation Results	118
6.1.1	Multiple Rank Problem	119
6.1.2	Multiple Point Location Problem	121
6.2	Conclusions and Future work	128
	BIBLIOGRAPHY	130

VITA

155

LIST OF FIGURES

FIGURE	PAGE
1.1 <i>A mesh with multiple broadcasting of size 4×4</i>	13
2.1 <i>The setting for Stage 1 of the generic algorithm</i>	17
2.2 <i>The Stage 1 of the generic algorithm</i>	18
2.3 <i>The Stage 2 of the generic algorithm</i>	19
2.4 <i>The Stage 2 of the generic algorithm</i>	21
2.5 <i>The Stage 3 of the generic algorithm</i>	22
2.6 <i>Query q can lie outside $CH(\text{subsets of } A)$ but lies within the $CH(A)$</i> .	29
2.7 <i>A wedge centered at q</i>	30
2.8 <i>Illustrating the proof of Lemma 2.5</i>	30
2.9 <i>The operation \diamond</i>	31
2.10 <i>The CLOSEST-SEGMENT problem</i>	36
2.11 <i>The MULTI-CIRCLE problem</i>	38
2.12 <i>The reduction of MULTI-CIRCLE to CLOSEST-SEGMENT</i>	39
2.13 <i>The MULTI-RANGE problem</i>	40
2.14 <i>An instance of the MULTI-STABBING problem</i>	44
2.15 <i>An instance of the POLY-LOCATION problem</i>	46
3.1 <i>Sorted and fully sorted matrices</i>	49
3.2 <i>The matrix view of the $ACM(n, p, M)$</i>	51

3.3	<i>The partition of matrix A</i>	52
3.4	<i>The sorted sequence $C_{i,j}$</i>	54
3.5	<i>Illustrating the proof of Lemma 3.4.</i>	56
3.6	<i>The concept of active copy of query q_k</i>	58
3.7	<i>The active copies of query q_k</i>	59
3.8	<i>Target of the gather operations in Stage 3</i>	64
4.1	<i>Adversary instance of the gather problem</i>	67
4.2	<i>Construction for multiple rank problem</i>	69
4.3	<i>Construction for multiple point location problem</i>	71
4.4	<i>Reduction for multiple point location problem</i>	72
4.5	<i>Reduction for multiple closest segment problem</i>	73
4.6	<i>Mapping for multiple stabbing problem</i>	75
4.7	<i>Partition into submeshes $R_{i,j}$</i>	77
4.8	<i>Data movement of Case 2</i>	80
4.9	<i>Processing of pure and impure query-rows</i>	84
4.10	<i>Stage 2 of MULTI-LOCATION</i>	87
5.1	<i>Lower bound for solving a single query</i>	100
5.2	<i>Input to the BSR problem</i>	102
5.3	<i>Partition into submeshes $R_{i,j}$</i>	103
5.4	<i>The data movement of Stage 1</i>	104
5.5	<i>Illustrating the proof of Lemma 5.10.</i>	107
5.6	<i>The concept of active copy of query q_k, for an MMB</i>	108
5.7	<i>Assignment of buses</i>	110
5.8	<i>Target of the data movement in Stage 3, for an MMB</i>	113
5.9	<i>Combining solutions, for an MMB</i>	114

5.10	<i>Partial solutions contained by processors in first column</i>	116
5.11	<i>Submeshes D_1, D_2, \dots, D_m</i>	117
6.1	<i>Running times for multiple rank problem</i>	120
6.2	<i>The best case running times for multiple rank problem</i>	121
6.3	<i>Multiple point location problem: case 1</i>	123
6.4	<i>Multiple point location problem: case 2</i>	124
6.5	<i>Multiple point location problem: case 3</i>	125
6.6	<i>Multiple point location problem: case 4</i>	126
6.7	<i>Running times for a network of workstations</i>	127

CHAPTER 1

INTRODUCTION

1.1 Overview

The maturation of computer science as a discipline has strengthened the need to consolidate isolated algorithms and techniques into general computational paradigms. The benefits of such an effort include the following:

- problems previously treated in isolation from one another can be shown to belong to the same class,
- once established, the paradigm will become a powerful tool, and
- effort involved in the implementation is reduced, owing to the uniformity offered by the paradigm.

By way of illustration, in a number of seemingly unrelated contexts in database design, pattern recognition, image processing, VLSI design, computer vision, and robot navigation, one is given collections A and Q of objects and a goal which is either to identify a collective property of the objects in $A \cup Q$, or to find for each object in Q a subset of A satisfying a given predicate. To further specify the illustration, consider the following examples. In virtual reality and computer

graphics, in the presence of a scene populated with a collection A of objects, a crucial problem is to identify the visibility horizon for a set Q of observers [35]. A somewhat different problem is of interest in path planning and collision avoidance problems in robotics [50] where navigational courses for a set Q of mobile robots is sought in the presence of a set A of obstacles. In pattern recognition, the well-known classification process involves comparing an unknown pattern Q against a template A and deciding whether the similarity measure is larger than a certain application-dependent threshold. In facility-location problems one is typically interested in an optimal placement of a set Q of facilities (schools, hospitals, etc.), amongst a set A of existing sites, in such a way that some constraints are satisfied [1, 75]. A similar problem arises in integrated circuit design in VLSI, where one is interested in the addition of a set Q of modules meant to enhance the functionality of the board A . In this latter context, it is customary to formulate the problem as a visibility problem involving collections A and Q of iso-oriented, non-overlapping, rectangles in the plane subject to a series of location constraints [66]. There are also some fundamental problems in computational geometry [1] which can be cast in the form of a object A and a set of queries Q , the answer is a combination of the solutions of all queries; such as, A and Q are convex polygons, determine if A and Q intersect.

All the problems mentioned informally above are traditionally solved using ad-hoc techniques developed by researchers within their respective fields. This dissertation combines all of these problems under a single umbrella, by providing a unified framework, of which the aforementioned problems will be instances. This will make it possible to provide uniform solutions to all these problems. As a first step, this work addresses the problems in the context of solving them in an abstract computational model. Next this work looks at the performance of the general solu-

tion on different computational models from both a theoretical and practical point of view. The unifying framework comes in the form of a generic problem referred to as the *Multiple Query* problem (MQ, for short). The main contribution of this work is to provide a computational paradigm which involves the MQ problem and a solution for the problem on abstract computational model. The power of the paradigm is demonstrated by obtaining time-optimal solutions to some problems on the mesh with multiple broadcasting.

To show the relevance of the paradigm a brief overview of some of the practical problems solved by the paradigm is presented in subsequent paragraphs. In robotics, objects are represented by convex hulls and operations on convex hulls are fundamental tools for various algorithms. The convex hull of a set, S of points in the plane, is the smallest convex set containing S . The convex hull is not only central to practical applications in robotics, but is a very useful tool for the solution of a number of questions arising in other areas of computer science, namely pattern recognition [3, 29], image processing [70, 71], and stock cutting and allocation [33, 34, 76]. In many applications, the problem of point location relative to a convex hull occurs quite frequently. Given a convex hull and a set of points in the plane determine for every point if it lies within the convex hull or not. The problem can be generalized, by asking the same question with respect to a simple polygon. Another application in robotics is, given a set of convex objects (obstacles) and set of points (different positions from which the robot views the obstacles), determine for each of the point, the range (two lines passing through the point) within which all the obstacles are located. Once determined this range enables the robot to stay clear of the obstacles.

Visibility is a fundamental problem in many areas of computer science. Given a set of line segments and a set of points in the plane, the visibility problem asks to determine, for each point, the segments which are closest to this point in the vertical direction. Recently, Bhagavathi *et al.* [11], and Gurla [39], have shown that visibility has many applications including triangulation. This problem has been extensively studied in various contexts and a variety of solutions exist in the literature. It can be shown that visibility problem is an instance of the general framework. Some search related problems also fall into this category. For example, given a set of points and a set of non-intersecting objects in the plane, for each object determine the number of points it contains. This problem is a direct application of the following scenario: identify all the branches of a corporation located within a given set of regions. This dissertation will also demonstrate that some problems in computational geometry such as the line stabbing [31] problem are instances of the MQ problem.

The MQ problem is sufficiently general to encompass a number of problems, and generally does not require the set of items A to have any structure. Frequently, a structured input leads to a faster algorithm, and some domains naturally offer structured inputs. In this dissertation, an example of one such problem domain will be seen, namely the *sorted matrices*. A matrix of elements is said to be sorted if both its rows and columns are independently sorted. A *fully sorted* matrix, sorted in either row major order or column major order, is a special case of the sorted matrix.

Sorted matrices provide a natural generalization of a number of real-life situations. Consider vectors $X = (x_1, x_2, \dots, x_n)$ and $Y = (y_1, y_2, \dots, y_n)$ with $x_i \leq x_j$ and $y_i \leq y_j$, whenever $i \leq j$. The Cartesian sum of X and Y , denoted $X + Y$ is the $n \times n$ matrix whose ij^{th} entry is $x_i + y_j$. The $X + Y$ matrix is sorted.

Searching, ranking and selection in sorted matrices are used in the development of fast algorithms in VLSI design, optimization, statistics, database design, and facility location problems and have received considerable attention in the literature [25, 26, 32, 37, 40, 58, 78].

Much of the theoretical work done in parallel algorithms, has focussed on the design and analysis of algorithms for the Parallel Random Access Machine (PRAM). The simple characteristics of PRAM make it suitable for developing theoretical results for evaluating the complexity of parallel algorithms. However, only a small number of real architectures (some bus-based multiprocessors like Encore and Sequent) can be considered conceptually similar in design with the PRAM model.

Although any real machine can simulate the PRAM model, it is nevertheless true that algorithms designed for network-based models will better match the architectures of existing parallel machines like Intel Paragon, Intel iPSC/860, CM-5, MasPar MP-1, IBM SP2, where processors with local memories are interconnected through a high-speed network supporting message-based communication.

The mesh-connected computer has emerged as one of the most widely investigated parallel models of computation. Mesh connected computers provide a natural platform for solving a large number of problems in computer graphics, image processing, robotics, and VLSI design. In addition, due to its simple and regular interconnection topology, the mesh is well suited for VLSI implementation [10]. The main problem with the mesh connected computer is its communication diameter, that is, if the data moved across the mesh takes $O(\sqrt{n})$ time, for a mesh of size $\sqrt{n} \times \sqrt{n}$. Frequently, the lower bounds and the running times of algorithms are dictated by the communication diameter of the mesh.

To overcome this problem, the mesh architecture has been enhanced by various types of bus systems [17, 43, 48, 52, 72, 80]. Early solutions, involving the addition of one or more global buses, shared by all the processors, have been implemented on a number of massively parallel machines [17]. Recently, a more powerful architecture, referred to as mesh with multiple broadcasting, has been obtained by adding one bus to every row and to every column of the mesh [43, 65]. The mesh with multiple broadcasting has proven to be feasible to implement in VLSI, and is used in the DAP family of computers [65]. Note that even here the problem of communication diameter comes to play for any class of problems which involves significant data movement. For example, if the problem requires rearrangement of its data, it generally takes \sqrt{n} time to do this task. But if the problem lends itself to “sparsification” (where the input size can be reduced by some processing) then better algorithms can be obtained.

Another computational model of theoretical interest as well as being commercially available is the mesh with multiple broadcasting. In recent years, efficient algorithms for solving a number of computational problems on meshes with multiple broadcasting have been proposed in the literature. These include image processing [44, 65], computational geometry [13, 14, 16, 43, 61, 63, 64], semigroup computations [8, 15, 19, 43], sorting [11], multiple-searching [13], and selection [12, 19, 43], among others.

With the advances in technology, diverse parallel computational models such as those described above continue to emerge. Each time a new model is introduced, considerable time and resources are invested to develop all the algorithms again from scratch. This difficulty can be tackled by designing algorithms for a general model, which in turn will enable them to be ported to many platforms. The abstract

computational model (ACM, for short) is one such model. An ACM is a multiprocessor system which consists of a set of primitive communication operations like broadcasting, and scatter where each operation has a cost associated to it. The cost is architecture-dependent, and reflects the amount of time taken by that operation relative to other operations on that architecture. This model was introduced in [39]. The remainder of the dissertation is organized as follows: the rest of Chapter 1 describes the state of the art and formalizes the various models of computation considered here, Chapter 2 contains algorithms for the MQ problem and its applications on the ACM model of computation, Chapter 3 adapts the algorithm for the Batched Searching and Ranking (BSR) problem on the ACM, Chapter 4 describes algorithms for the MQ problem and its applications on the MMB and details the lower bounds achieved there, Chapter 5 describes a time-optimal algorithm for the BSR problem on the MMB, and finally, Chapter 6 contains the conclusions along with the implementation results and pointers for future work.

1.2 State of the Art

The idea of a general framework for the problems mentioned in the previous section has not been seen before; however, many of the problems have been addressed previously on diverse computational models. This section presents the current running times for these solutions as found in the literature. These problems fall into three broad categories multiple search problems, visibility related problems, and proximity problems.

The multiple search problem can be stated as follows: given a sorted sequence, A , of items and a set of queries Q , for each query, $q \in Q$, determine the position of the largest element of A less than or equal to q , let $|Q| = m$ and

$|A|=n$. A lot of work has been done on the multiple search problem, which focussed mainly on solving it, and sometimes using that solution to solve some applications. Akl and Meijer [2] first presented a parallel solution to this problem where they solved the problem on a m processor EREW PRAM with a running time of $O(\frac{\log m \log n}{\log \log n})$. A faster solution was presented by Wen [85], with a running time of $O(\log m + \log n)$. Chao *et al.* [18] provide a solution to the multiple search problem in on a three dimensional reconfigurable mesh. They solve the problem in $O(1)$ time on a $n^{\frac{1}{2}} \times n^{\frac{1}{2}} \times n^{\frac{1}{2}}$ reconfigurable mesh. Finally, Bhagavathi *et al.* [13] provide a solution to the multiple search problem on an enhanced mesh. Here the problem is solved on a $n^{\frac{1}{2}} \times n^{\frac{1}{2}}$ MMB in $O(m^{\frac{1}{2}})$ time. Until now there has been no effort to provide a unifying framework. It will be demonstrated that the multiple search problem is a particular case of an instance of the general problem, the input A need not be completely sorted.

Reif and Sen [68] present a randomized parallel algorithm for the point location problem with n queries which takes $O(\log n)$ time with high probability on a CRCW PRAM. Further work on this problem for the PRAM has been done in [4, 38, 81]. This problem has been solved by Chazelle [21], on a linear array with k query points in $O(k + n)$, where n is the size of the input data. On a mesh, Jeong and Lee [42] solve the problem in $O(n^{\frac{1}{2}})$. In [28], Dehne solved the separability problem in $O(n^{\frac{1}{2}})$ time on a mesh of size n . Finally, Sarkar and Stojmenović [77] using a CREW PRAM of n processors solve the problem in $O(\log n)$ time.

The visibility problem has been solved by Atallah *et al.* [5] in $O(\log n \log \log n)$ time using $O(n)$ processors on a CREW PRAM. The result was later improved by Atallah *et al.* [4] to $O(\log n)$ time using $O(n)$ processors on a CREW PRAM. A randomized algorithm was presented by Reif and Sen in [68]. Atallah and Tsay [6]

give an algorithm to solve the visibility problem on a linear array of size N which runs in $O(n \frac{\log n}{\log N})$ time, where $N < n$. On a hypercube the problem is solved in $O(\log n \log \log n)$ time by MacKenzie and Stout [57]. A randomized algorithm is presented by Reif and Sen [69] which runs in $O(\log n)$ probabilistic time on an $O(n)$ processor butterfly.

Solutions to the proximity problems for the mesh and a linear array, with $O(n)$ processors, are described in [21, 55], which require $O(n^{\frac{1}{2}})$ and $O(n)$ time, respectively. In [79], Steiger and Streinu show that proximity problem can be solved in $O(\log^2 n)$ time on a $O(n)$ processor hypercube. Using the algorithm, in [57] MacKenzie and Stout show the running time on a hypercube of size n is $O(\log n (\log \log n)^2)$. In the PRAM, the proximity problems were solved in $O(\log n)$ time by Cole and Goodrich [22] on an n processor CREW PRAM; they also achieved the same running time on an EREW PRAM with an increase in the memory size by a factor of $O(\log n)$. Histogram computation of a digital image is a classic image processing problem which has been looked at by many researchers. In [62], Olariu *et al.* solve the problem on a reconfigurable mesh of size $n \times n$ in $O(\log \log n)$ time. In [44], Prasanna and Reisis solve the problem on an MMB.

1.3 Models of Computation

In this section, a trace of the different parallel models of computation is presented, followed by a brief discussion of the models employed in this dissertation. The early models of computation included *Perceptrons*, proposed in the late 1950's [73] and Cellular Automata [23]. Then came the interconnection networks like the linear arrays, meshes or two-dimensional arrays, several variations of meshes including the meshes with broadcast buses referred to as meshes with multiple broadcasting,

and the meshes with reconfigurable buses. Tree networks, mesh-of-trees, pyramid networks, hypercube, cube-connected cycles, butterfly, AKS sorting network, star and pancakes are among the other network based models of computation which have been studied. Shared memory models of computation include PRAMs, scan model, broadcasting with selective reduction. For a description of network based and shared memory models refer to [1]. Recent models like Valiants BSP model [84] and the LogP model [27] take into account the communication costs by introducing network related parameters (e.g., latency) into the model. In this dissertation, the following platforms are used, ACM, MMB, and IBM-SP2. A discussion of each of these models follows.

In this dissertation, the model of computation has to encompass a wide range of parallel models from fine grained to coarse grained models. Also the communication primitives need to be fairly high level. The Abstract Computational Model [39] meets the requirements and can be characterized as follows.

An $ACM(n, p, M)$ has p processors, each processor with memory of size $O(M)$, $M > \frac{n}{p}$ where n is the maximum input data size. All the processors are assumed to be identical and are enumerated as P_0, P_1, \dots, P_{p-1} . Each processor P_i knows its identity i . The computational power of a processor is assumed to be directly proportional to the size of the memory $O(M)$. This assumption will facilitate the unification of coarse grain and fine grain multiprocessor systems. The local operations performed by the processors vary from very simple to very complex depending upon the number of processors. In the fine grain scenario where there may be a large number of processors, each processor is capable of simple arithmetic and communication operations. In the coarse grain scenario, there are a small number of powerful processors. Communication is done via an interconnection network. A

more detailed discussion can be found in [39].

The high level communication primitives of an $ACM(n, p, M)$ follow.

- *Broadcast*: A processor informs every other processor, of an $ACM(n, p, M)$, of some data, of size k . The time associated with a broadcast operation is $T_B(k, p)$.
- *Multicast*: A processor communicates a message (of size k) to a subset of processors (of size p') of an $ACM(n, p, M)$. The time associated with such a multicast operation is $T_M(k, p')$. Note that, it is possible to have parallel multicast operations among mutually exclusive processor subsets.
- *Point-to-Point*: A processor, P_i , communicates a message (of size k) to a processor, P_j , of an $ACM(n, p, M)$. The time associated with such a point-to-point operation is $T_P(k)$. Note that, it is possible to have parallel point-to-point operations among mutually exclusive processor subsets.
- *Reduce*: Perform an operation (e.g., sum, product) on p elements to give a single result where each processor contributes one element for the operation. This operation gives k results if each processor contributes k elements. In an $ACM(n, p, M)$, $T_R(k, p)$ represents the time for a reduction operation involving k elements per processor. The reduction of k elements over p' processors, is represented as $T_R(k, p')$.
- *Gather*: Collect data from selected set of processors. Specifically, if the operation involves a total of k elements and p' processors on an $ACM(n, p, M)$, $T_G(k, p')$ represents the time for the operation. When the operation is complete, the processor issuing the gather operation will have collected k elements.
- *All-to-All Gather*: This operation is similar to the gather but all the processors will receive some data. Specifically, if the operation involves k elements per processor and p' processors on an $ACM(n, p, M)$, let the processors involved in the operation be enumerated as $P_1, P_2, \dots, P_{p'}$. At the end of the operation P_1 will receive first

$\frac{k}{p'}$ elements from each processor. Similarly, P_2 will receive second set of $\frac{k}{p'}$ elements from each processor. It is the same for all the other processors. Let $T_{AAG}(k * l, p')$ represent the time for the operation. It has to be noted that this operation is usually quite expensive.

- *Scatter*: This is the reverse operation of gather, that is, data is to be distributed into various processors. Specifically, if the operation involves a total of k elements and p' processors on an $ACM(n, p, M)$, $T_S(k, p')$ represents the time for the operation.

An example of coarse grain machine is the IBM-SP2, it is built using powerful RS/6000 processors, an RS/6000 processor powerful enough to be used in a workstation. The communication medium of an IBM-SP2 is a switch (multi-stage omega network).

Consider next, the models related to mesh based computers. Mesh connected computers considered here provide insight for adapting the algorithms to fine grain machines. The mesh connected computer of size $M \times N$ is a machine with MN processors arranged in rectangular array. The processor $P(i, j)$, representing the processor in row i and column j and is connected via bi-directional unit-time communication links to its four *neighbors*, provided they exist. Each processor has a fixed number of registers, of size $O(\log MN)$ each and operates in SIMD mode: in each time unit, the same instruction is executed by all the processors. Each processor is assumed to know its own coordinates within the mesh. It is also assumed that a processor can perform standard arithmetic and boolean operations on the contents of its registers in $O(1)$ time.

Compared to other parallel architectures, meshes have the advantage that several already exist [9, 10, 41] and that their simple near-neighbor wiring allows

them to be constructed more economically than say, hypercubes. Its regular interconnection topology makes the mesh ideal for number of problems in geometry, image processing and graphics [30, 51, 54, 55, 56, 80].

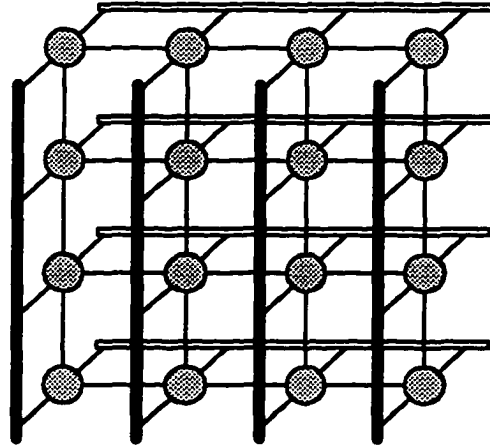


Figure 1.1: *A mesh with multiple broadcasting of size 4×4*

A mesh with multiple broadcasting, MMB, of size $M \times N$ consists of MN identical processors positioned in a rectangular array overlaid with a bus system, refer to Figure 1.1. In every row of the mesh the processors are connected to a horizontal bus; similarly, in every column the processors are connected to a vertical bus. To keep the model realistic, only one processor is allowed to broadcast on a given bus at any one time. By contrast, all the processors on the bus can simultaneously read the value being broadcast. In accord with other researchers [8, 17, 43, 48, 65], it is assumed that communications along buses take $O(1)$ time. Although inexact, recent experiments with the DAP, the GCN, and the YUPPIE multiprocessor array systems seem to indicate that this is a reasonable working hypothesis [48, 65]. An MMB of size $\sqrt{N} \times \sqrt{N}$ can be viewed as an instance of the $ACM(n, p, M)$, here $n = p = N$, and $M = 1$, with the communication medium being mesh connections

along with row and column bus connections.

CHAPTER 2

THE COMPUTATIONAL PARADIGM ON THE ACM

In Chapter 1 a brief introduction of the proposed paradigm was sketched. The first main goal of this chapter is to present the paradigm in full detail on the Abstract Computation Model. This involves a formal definition of the MQ problem and a generic solution of the problem on the $ACM(n, p, M)$ which acts as a framework for other solutions. The second main goal of this chapter is to prove the power of the paradigm by demonstrating that many problems can be formulated as instances of the framework. Once the formulation is obtained the generic solution can then be customized to obtain solutions for individual problems. The following sections will then describe the process of formulation of the problems and the customization of the generic solution.

The remainder of the chapter is organized as follows. Section 2.1 offers a generic algorithm for the MQ problem. The remaining sections discuss various instances of the MQ problem. Specifically, Section 2.2 discusses rank-related problems; Section 2.3 discusses the multiple point location problem and several of its variants and applications; Section 2.4 addresses proximity-related problems; finally, Section 2.5 discusses the multiple stabbing problem.

2.1 A Generic Multiple Query Algorithm

A generic instance of the MQ problem has the following parameters:

- an arbitrary set $A = \{a_1, a_2, \dots, a_n\}$ of items
- an arbitrary set $Q = \{q_1, q_2, \dots, q_m\}$ of queries
- a decision problem $\phi : Q \times A \rightarrow \{\text{"yes"}, \text{"no"}\}$
- an associative and commutative function f operating on subsets of A

For every query q_i ($1 \leq i \leq m$), let $S_i = \{a_j \in A \mid \phi(q_i, a_j) = \text{"yes"}\}$. In this context, the *solution* of q_i is $f(S_i)$. It is noted that f generally acts more like an operator than a function in the strict sense of the word. The function f is commutative in the following way, $f(S^1 \cup S^2) = f(S^1) \otimes f(S^2) = f(S^2) \otimes f(S^1)$, where \otimes is determined by f and ϕ . Similarly, f is associative implies $f(S^1 \cup S^2 \cup S^3) = (f(S^1) \otimes f(S^2)) \otimes f(S^3) = f(S^1) \otimes (f(S^2) \otimes f(S^3))$.

The set A is stored in some order, $\frac{n}{p}$ items[2]* per processor, in an $\text{ACM}(n, p, M)$. The set Q is stored in the first $\frac{m}{M}$ processors ($P_0, P_1, \dots, P_{\frac{m}{M}-1}$), M queries per processor. Note that each processor can hold $O(M)$ elements, so the first $\frac{m}{M}$ processors can hold $\frac{n}{p} + M \leq 2M$ elements each. Throughout this chapter the layout of items and queries is assumed to be in the above format.

To make the notation less cumbersome, write[†]

$$s = \frac{m}{M} \tag{2.1}$$

*In an $\text{ACM}(n, p, M)$, $\frac{n}{p} \leq M$.

†For simplicity assume that s and $\frac{p}{s}$ are integers.

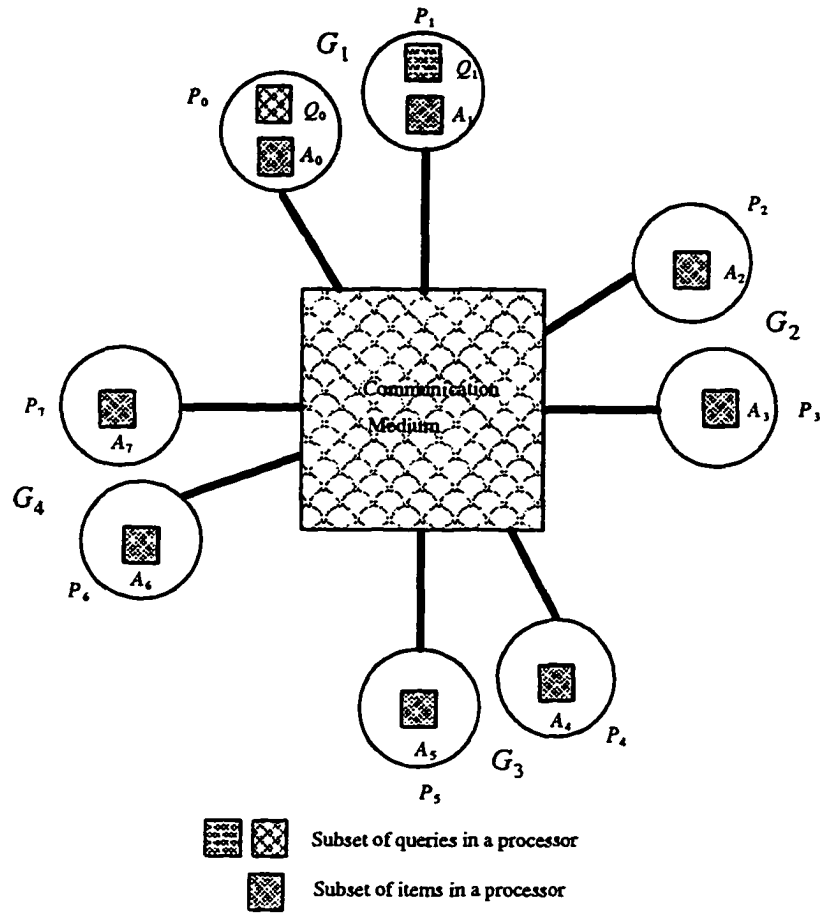


Figure 2.1: *The setting for Stage 1 of the generic algorithm*

In this notation, the $\text{ACM}(n, p, M)$ is viewed as consisting of $\frac{p}{s}$ groups $G_1, G_2, \dots, G_{\frac{p}{s}}$, where each G_i is an ACM with processors $P_{(i-1)s}, P_{(i-1)s+1}, \dots, P_{is-1}$, as illustrated in Figure 2.1. The number of processors chosen per group plays an important role in obtaining fast algorithms. In this chapter, for simplicity of exposition the number of processors per group is assumed to be $\frac{m}{M}$. The optimal choice, of the number of processors per group, depends upon the balancing of running times of different Stages of the algorithm. This will be demonstrated in Chapter 4 and Chapter 5.

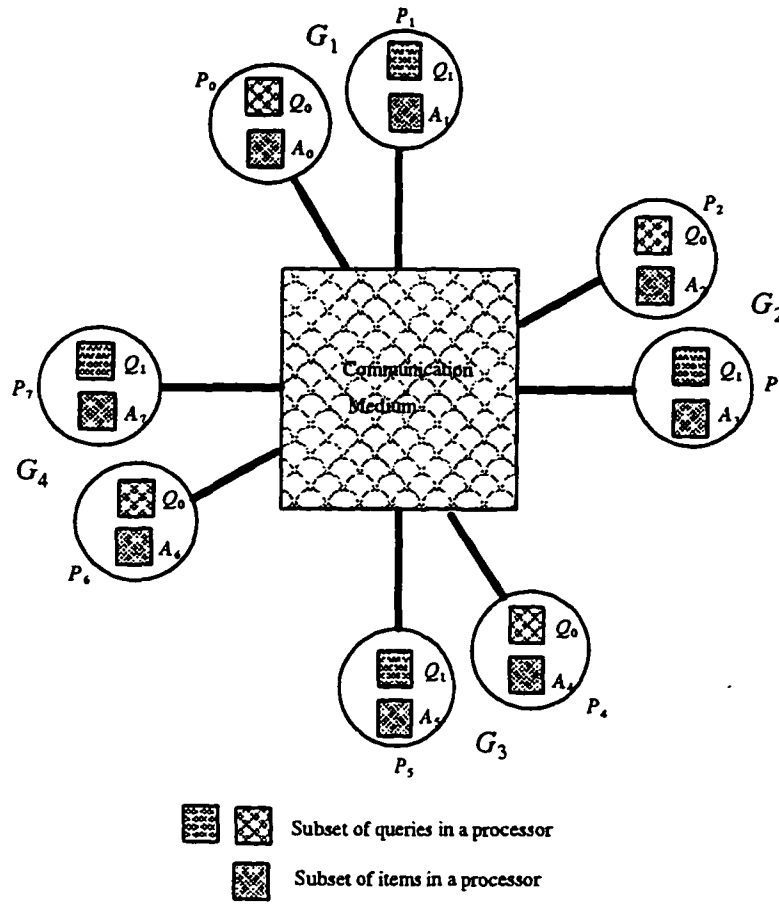


Figure 2.2: *The Stage 1 of the generic algorithm*

The generic algorithm for MQ problem consists of three distinct stages that are summarized as follows:

Stage 1. The goal of this stage is to replicate the set Q , stored in the group G_1 (the first s processors), refer to Figure 2.1, to all the other groups. To be more specific, in this stage each processor P_i , $0 \leq i \leq s - 1$, of G_1 will multicast the queries it contains to the corresponding processors in all the other groups; i.e., P_i will multicast to the processors P_{j*s+i} , $1 \leq j \leq \frac{p}{s} - 1$. Here the multicasts are done in parallel. It is important to note that, at the end of Stage 1, having replicated the queries, the

original instance of the MQ problem is partitioned into several instances, each local to a G_i . Every local instance involves the subset of A stored by the processors in G_i , the set Q of queries, a decision problem, and a function f . Figures 2.1, and 2.2 illustrate the data replication of this stage, here $G_1 = \{P_0, P_1\}$, $p = 8$, $s = 2$.

Stage 2. The principle goal of this stage is to solve in each group, G_i , the local

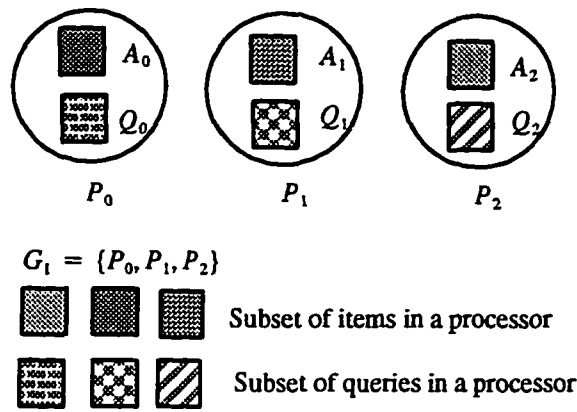


Figure 2.3: *The Stage 2 of the generic algorithm*

instance of the MQ problem. This will be done in parallel for all the groups. As the processing done in each group is similar, the operations performed in one group (G_1) will be described without loss of generality. The subset of items, of A , contained in processor P_i , $0 \leq i \leq s-1$ of G_1 , will be represented by A_i . Similarly, the subset of Q in P_i will be referred to as Q_i , refer to Figure 2.3. For a query to find its local solution it should “visit” all the items in G_1 . To achieve this goal, queries and items will perform computations and then the items will be passed across the processors in a cyclic fashion. This is referred to as the *compute-and-move* operation. To elaborate, consider the subset of items, A_i , present initially in P_i . In the j^{th} compute-and-move operation, $1 \leq j \leq s-1$, items in A_i will be located in processor $P_{(i+j-1) \bmod s}$. The

subset of queries $Q_{(i+j-1) \bmod s}$ present in processor $P_{(i+j-1) \bmod s}$ will perform the required computations with the A_i . At the end of this computation the partial solution that is associated with each query, $q \in Q_{(i+j-1) \bmod s}$, will be updated accordingly. Finally, the A_i will be moved to processor $P_{(i+j) \bmod s}$. After this, the $j + 1^{st}$ compute-and-move operation will begin. This will be done in parallel for all the items in G_1 . In Figure 2.4, (a), (b) and (c) illustrate three compute-and-move operations. Here there are three processors in the group. In the last compute-and-move operation the items need not be moved any more only the computation is required.

It may be intuitive to move the queries instead of the items. However, $M \geq \frac{n}{p}$ implies that communication costs will be less if items are moved.

Stage 3. The goal of this stage is to combine the solutions of the local instances of the MQ problem obtained in Stage 2 to get the global solution of the MQ problem. This involves s parallel reduce operations. Specifically, each processor P_i , $0 \leq i \leq s - 1$, of G_1 will perform a reduce operation with corresponding processors in all the other groups; that is, P_i will be involved in a reduce operation with processors P_{j*s+i} , $1 \leq j \leq \frac{n}{s} - 1$. Figure 2.5 depicts this reduce operations of Stage 3.

The running times of Stage 1 and Stage 3 are of the order of $T_M(M, \frac{n}{s})$ and $T_R(M, \frac{n}{s})$, respectively. This is independent of the problem being solved. In Stage 2, the compute-and-move operation can be made more efficient by overlapping the computation with communication. The following is a brief description of the processing and the running times involved in such an overlap. In a compute-and-move operation, as soon as a processor receives all the items for the current computation, it makes a copy of the items and put them in send buffer to the next processor. While the communication of the copy (of items) is taking place local computation will be

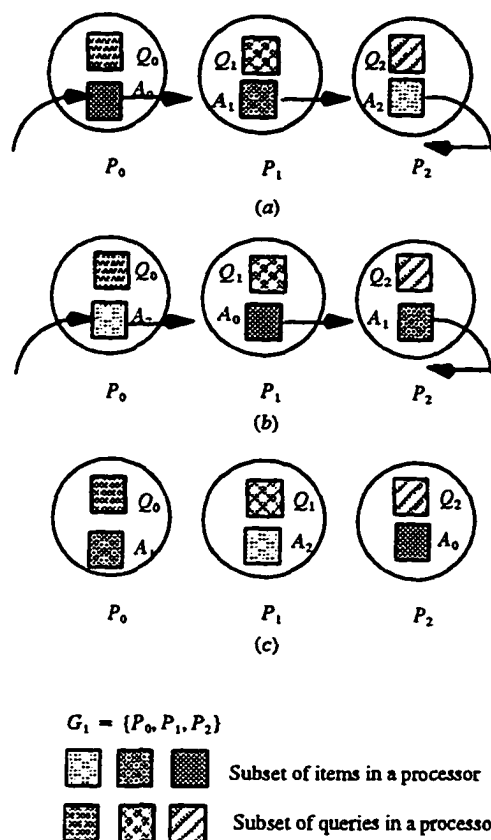


Figure 2.4: *The Stage 2 of the generic algorithm*

in progress. The communication time for this data movement will be $(s-1) * T_P(\frac{n}{p})$, and clearly, the computation time will depend on the problem being solved.

The purpose of the remaining sections of this chapter is to show that the MQ problem has many, and sometimes unexpected, applications to problems in database design, pattern recognition, image processing, robotics, and morphology. Each of the subsequent sections is typically organized as follows, first the statement of the problem being solved, followed by the formulation of the problem as an instance of the MQ problem, finally the details of the solution. All the algorithms for particular applications will involve fleshing out the processing in Stage 2, which is application-

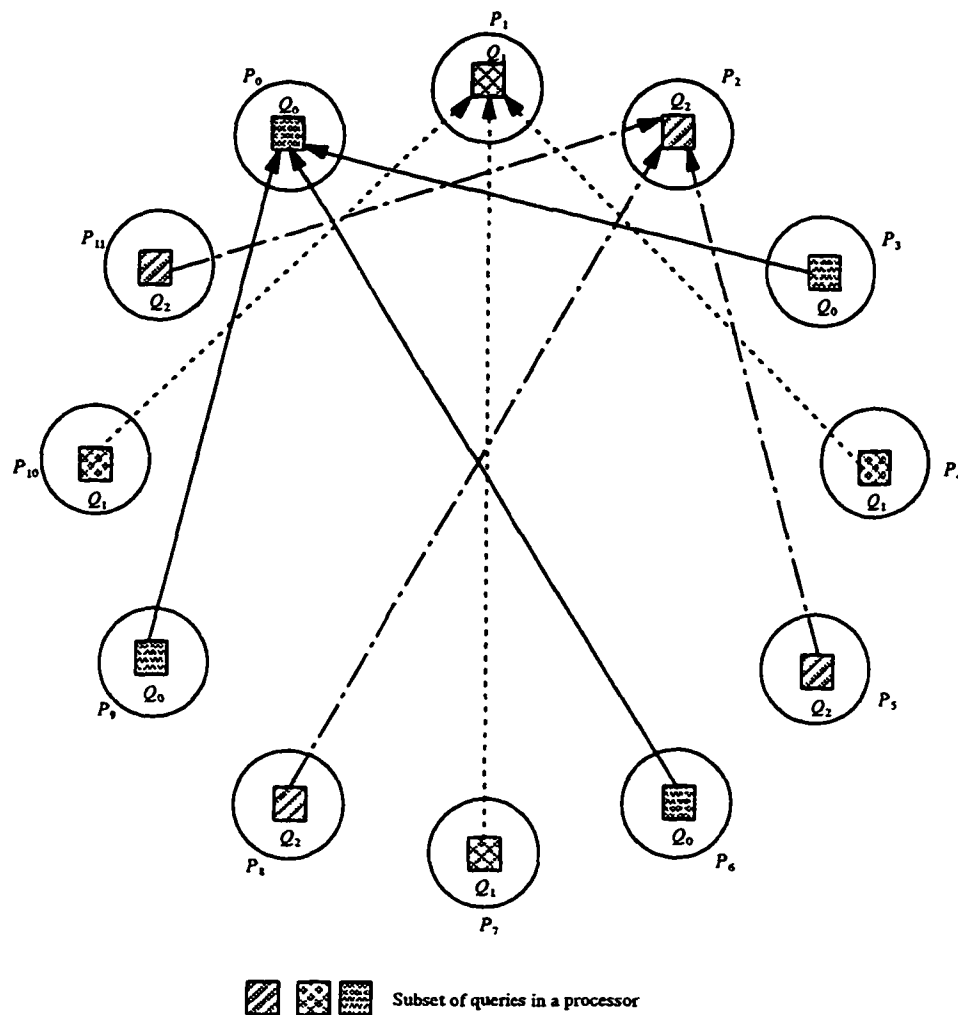


Figure 2.5: *The Stage 3 of the generic algorithm*

dependent. In each case the particular function f required for the formulation will be easily seen to be both associative and commutative, in such a way that the generic processing of Stage 3 will apply with minor changes. With these considerations, the complete algorithm will be presented in Section 2.2. In the remaining sections only the processing of one compute-and-move operation of Stage 2 will be described in detail. The details of Stage 3 will be presented only when the tasks involved are non-trivial.

2.2 Rank-Related Computations

The purpose of this section is to show that two fundamental problems in geographic data processing, database design, and image processing can be solved simply and elegantly by the paradigm by formulating them as the instances of the MQ problem.

2.2.1 The Multiple Rank Problem

Given a collection of items in a database along with a set of values, the *multiple rank* problem, is to compute for each query the number of items in the database that are smaller [2, 53]. The multiple rank problem is considered to be a fundamental algorithmic problem that finds additional applications in geographic data processing, computer graphics, image processing, computer vision, and morphology, to name just a few [2, 74, 85]. Akl and Meijer [2] as well as Wen [85] have studied the multiple rank problem in the PRAM model of computation. A simple variant of the multiple rank problem was solved in [13]. The multiple rank problem will be referred to as MULTI-RANK. It will be shown that it can be stated as a multiple query problem.

For definiteness, both the items in the database and the set of values are assumed to come from a totally ordered universe. The corresponding instance of the MQ problem has the following parameters:

- the set $A = \{a_1, a_2, \dots, a_n\}$ of items,
- the set $Q = \{q_1, q_2, \dots, q_m\}$ is the set of values,
- the decision problem $\phi : Q \times A \rightarrow \{\text{"yes"}, \text{"no"}\}$ is such that $\phi(q_i, a_j) = \text{"yes"}$ whenever $a_j < q_i$,

- $f(S) = |S|$.

For every i ($1 \leq i \leq m$), let S_i be the set of items a_j in A for which $\phi(q_i, a_j) = \text{"yes"}$. The solution to query q_i is $f(S_i)$, in other words it is *rank* of q_i among the items in A .

The algorithm for MULTI-RANK consists of the following stages.

Stage 1. Replicate the set Q as in the Stage 1 of the generic algorithm.

Recall that, the $\text{ACM}(n, p, M)$ is viewed as consisting of $\frac{p}{s}$ groups $G_1, G_2, \dots, G_{\frac{p}{s}}$, where each G_i is an ACM with processors $P_{(i-1)s}, P_{(i-1)s+1}, \dots, P_{is-1}$, where $s = \frac{m}{M}$. After the replication each G_i contains the query set Q . At the end of this stage the instance of the original MQ problem is partitioned into several instances, each local to a group.

Stage 2. As in the Stage 2 of the generic algorithm, the operations performed in group G_1 will be presented. Similar operations are performed in all the other groups, in parallel. As an initialization step, subset of items in every processor are sorted in increasing order. The processing done within a processor P_i in G_1 , in one compute-and-move operation is as follows. To simplify the notation, let $b_1, b_2, \dots, b_{\frac{n}{p}}$ stand for the sorted sequence of items in processor P_i . A value l_1 is associated with each query to store the final rank of the query, the initial value of l_1 is zero. Each query will do a binary search to find out its rank among $b_1, b_2, \dots, b_{\frac{n}{p}}$. This rank is added to the l_1 . Now the sorted set of items, $b_1, b_2, \dots, b_{\frac{n}{p}}$ will be moved on to the next processor and the next compute-and-move operation begins. This processing is done in parallel for all the item subsets within G_1 .

The running time for this stage is dictated by the sorting operation, the binary search, and the communication of items; the cost of each of these operations is $O(\frac{n}{p} \log \frac{n}{p})$, $O(m \log \frac{n}{p})$, and $O(T_P(\frac{n}{p}))$, respectively. Each compute-and-

move operation costs $O(m \log \frac{n}{p} + T_P(\frac{n}{p}))$ so the total $s - 1$ operations will cost $O(m \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p})))$. The result is summarized as follows:

Lemma 2.1. The task of solving the queries in every G_i can be performed in $O((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p})))$ time. \square

Stage 3. At the end of Stage 2, every processor of every group G_i that stores a query q will also store its local solution l_i . The goal of Stage 3 is to compute the sum $l_1 + l_2 + \dots + l_s$, for every query $q \in Q$. This task can be carried out as in Stage 3 of the generic algorithm discussed in the previous section.

Stage 1 and Stage 3 have running times of $O(T_M(M, \frac{p \cdot M}{m}))$ and $O(T_R(M, \frac{p \cdot M}{m}))$, respectively. Consequently the following result is stated.

Theorem 2.2. An arbitrary instance of the MULTI-RANK problem involving a set of n items and a set of m queries can be solved in $O(T_M(M, \frac{p \cdot M}{m}) + ((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p \cdot M}{m}))$ on an $ACM(n, p, M)$. \square

2.2.2 Histogram Computation

The task of computing the histogram of a gray-level image is one of the fundamental operations in pattern recognition and low-level vision [7, 29]. The goal of this subsection is to show that the histogram computation problem can be formulated as a MQ problem. Further, a simple and elegant solution on an $ACM(n, p, M)$, will be obtained by using the algorithm for MULTI-RANK as a subroutine.

Let A be a gray-level digital image of size $\sqrt{n} \times \sqrt{n}$ pretiled onto an $ACM(n, p, M)$ $\frac{n}{p}$ pixels per processor. Assuming that the gray-scale involves m values, the goal is to compute the histogram of the given image. This problem is referred to as HISTOGRAM. The corresponding instance of the MQ problem has the following parameters:

- the set $A = \{a_1, a_2, \dots, a_n\}$ is the set of pixels in the given gray-level image,
- the set $Q = \{q_1, q_2, \dots, q_m\}$ consists of m gray-level intensities,
- the decision problem $\phi : Q \times A \rightarrow \{\text{"yes"}, \text{"no"}\}$ is such that for every $1 \leq i \leq m$ and $1 \leq j \leq n$, $\phi(q_i, a_j) = \text{"yes"}$ if and only if $q_i = a_j$,
- $f(S) = |S|$.

For every i ($1 \leq i \leq m$), let S_i be the set of items a_j in A for which $\phi(q_i, a_j) = \text{"yes"}$. The solution to query q_i is $f(S_i)$ which is its frequency in the given image.

The algorithm for HISTOGRAM is identical to the algorithm for MULTI-RANK except for a post-processing step that is now described. Let $\text{rank}(q_1), \text{rank}(q_2), \dots, \text{rank}(q_m)$ be the ranks of the queries returned by MULTI-RANK applied to the instance of the HISTOGRAM problem. Now for every i ($1 \leq i \leq m-1$) the solution to q_i is $\text{rank}(q_{i+1}) - \text{rank}(q_i)$, in other words, the number of pixels in A having a gray-level intensity equal to q_i . Furthermore, the solution to q_m is $n - \text{rank}(q_{m-1})$. The running time for HISTOGRAM will be the of the same order as the MULTI-RANK. It is noted that the solution for HISTOGRAM can be obtained without using the MULTI-RANK as a subroutine, but by using the same algorithm with some minor variations in Stage 2. That is instead of finding the ranks each query will determine the number of items with the same value. To summarize the findings the following result is stated.

Theorem 2.3. An arbitrary instance of the HISTOGRAM problem involving an m -level image of size n can be solved in $O(T_M(M, \frac{p \cdot M}{m}) + ((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p \cdot M}{m}))$ on an $\text{ACM}(n, p, M)$. \square

2.3 The Multiple Point Location Problem

The purpose of this subsection is to show an elegant solution to the multiple point location problem by reducing it to an instance of the MQ problem. Further, it will be shown that the multiple point location problem itself has some applications. Just like the classic point location problem, the multiple point location problem is central to computer graphics, pattern recognition, image processing, robotics, and morphology [1, 7, 29, 35, 50, 75]. Let $A = \{a_1, a_2, \dots, a_n\}$ and $Q = \{q_1, q_2, \dots, q_m\}$ ($1 \leq m \leq n$) be arbitrary sets of points in the plane. The points in Q will be referred to as *query points*. The *multiple point location problem*, (MULTI-LOCATION, for short) is to determine for every subscript i ($1 \leq i \leq m$) whether the query point q_i lies inside the convex hull $\text{CH}(A)$ of A . Without loss of generality, the points are assumed to be in general position.

The layout of the points of A and Q is the same as described in the Section 2.1. Before solving the problem, some geometric preliminaries will be discussed. Recall that if a point q is exterior to $\text{CH}(A)$, then there exist exactly two supporting lines from q to $\text{CH}(A)$. In fact, the converse is also true: a point q lies to the exterior of $\text{CH}(A)$ if there exist supporting lines from q to $\text{CH}(A)$ (refer to Figure 2.7). Let P be a convex polygon and let q be a point outside P . A supporting line δ from q to P will be termed a *left support line* if P lies in the right halfplane determined by assigning δ the direction away from q and towards P . Otherwise, δ will be termed a *right support line*.

As it turns out, the MULTI-LOCATION problem can be stated as an instance of the MQ problem with the following parameters:

- a set $A = \{a_1, a_2, \dots, a_n\}$ of items which is precisely the given set of points in the plane,

- a set $Q = \{q_1, q_2, \dots, q_m\}$ of queries consisting of the m query-points,
- a decision problem $\phi : Q \times A \rightarrow \{\text{"yes"}, \text{"no"}\}$ such that $\phi(q_i, a_j) = \text{"yes"}$ if and only if the line determined by q_i and a_j is a supporting line for $\text{CH}(A)$,
- $f(S) = S$.

The function f implies that the solution for each query is the supporting lines, if they exist.

The task specific to Stage 2 is to determine for every point in Q whether it is interior to any of the convex hulls local to a group. Clearly, a query point that is interior to any such convex hull lies in the interior of the convex hull of A and its corresponding solution is the empty set. As a technicality, for all i ($1 \leq i \leq m$), S_i is initialized to the empty set. The compute-and-move operation of the MULTI-LOCATION algorithm is as follows: only the details for group G_1 are presented.

As an initialization step the convex hull of the subset A_i of A is computed. This task can be performed in $O(\frac{n}{p} \log \frac{n}{p})$ time using an optimal sequential convex hull algorithm [67]. For simplicity of exposition, assume that the convex hull of A_i is the convex polygon $C = c_1, c_2, \dots, c_{\frac{n}{p}}$.

A pair of tangents (l, r) is associated with each query q . These are the tangents from q to the convex hull of A' . Here A' is the set of all items that the query q has encountered in all the previous compute-and-move operations. In the current compute-and-move operation in processor P_j each query q will determine a pair of tangents to the convex hull of the subset A_i of items currently stored by P_j . The pair (l, r) is updated with the newly computed set of tangents. At the end of this operation A_i is moved to the next processor of the group. The process of updating of (l, r) is a non-trivial task and is a part of Stage 3 of this algorithm. A

better perspective of the operation is obtained by looking at the complete processing of Stage 3.

Consider the running time of this stage: finding tangent to a convex hull is a variant of the binary search [67], and it takes logarithmic time. Consequently the following result is stated.

Lemma 2.4. For every query point in Q , the supporting lines to the convex hull of the subset of A in each G_i can be found in $O((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p})))$ time. \square

Stage 3. The main goal of this stage is to use the information obtained in Stage 2 of the algorithm to decide which query points lie in the interior of $CH(A)$. It is important to note that for a point q to lie outside of the convex hull of A it is not sufficient that q lie outside of the subset of A in all the G_i s. Figure 2.6 illustrates the situation: the point q lies outside of the three convex hulls, but not outside their union.

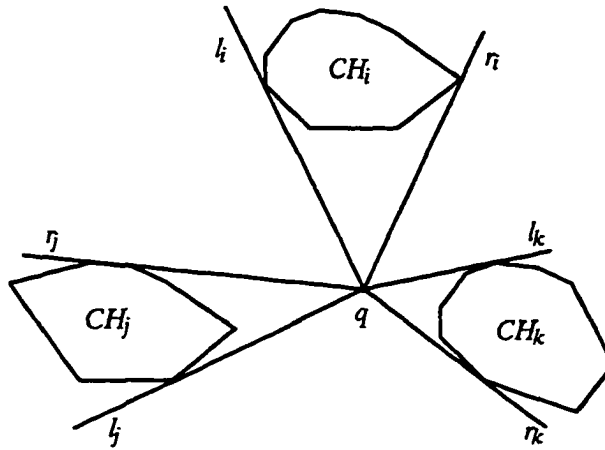
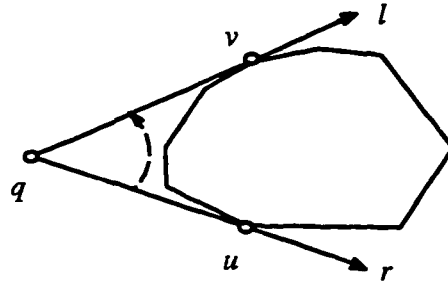


Figure 2.6: Query q can lie outside $CH(\text{subsets of } A)$ but lies within the $CH(A)$

For every query point q lying outside of the convex hull of the points in

Figure 2.7: A wedge centered at q

some generic processor P_i , the information obtained in Stage 2 is perceived as a solid wedge centered at q . This wedge is specified, in counter-clockwise order, as an ordered triple (u, q, v) such that \overrightarrow{qu} and \overrightarrow{qv} are the right and left supporting rays from q to the corresponding convex hull. For convenience, \overrightarrow{qu} and \overrightarrow{qv} are referred to as r and l , respectively. When this happens, the wedge (u, q, v) will be specified as (r, q, l) . Figure 2.7 illustrates this concept.

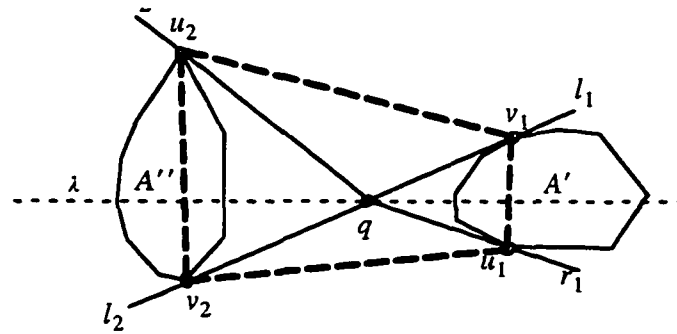


Figure 2.8: Illustrating the proof of Lemma 2.5

Now assume that the solutions for the same query q in two sets of points, A' and A'' , are to be combined: by the above discussion, these solutions are planar wedges, centered at q , specified as ordered triples (u_1, q, v_1) and (u_2, q, v_2) , consisting of the right and left supporting rays, respectively, from q to the convex hull of A'

and A'' . The following technical result is key to Stage 3 of the algorithm.

Lemma 2.5. If there exists a line passing through q and intersects both (u_1, q, v_1) and (u_2, q, v_2) then q lies in the interior to the convex hull of $A' \cup A''$.

Proof. Let λ be the line that passes through q and intersects both (u_1, q, v_1) and (u_2, q, v_2) , refer to Figure 2.8. This assumption guarantees that the pairs of points (v_1, u_2) and (v_2, u_1) lie in opposite halfplanes determined by λ . In turn, this guarantees that q lies inside the convex hull of the points u_1, v_1, u_2, v_2 . Now, a well-known result of Yaglom [86] guarantees that q lies inside the convex hull of A' and A'' . \square

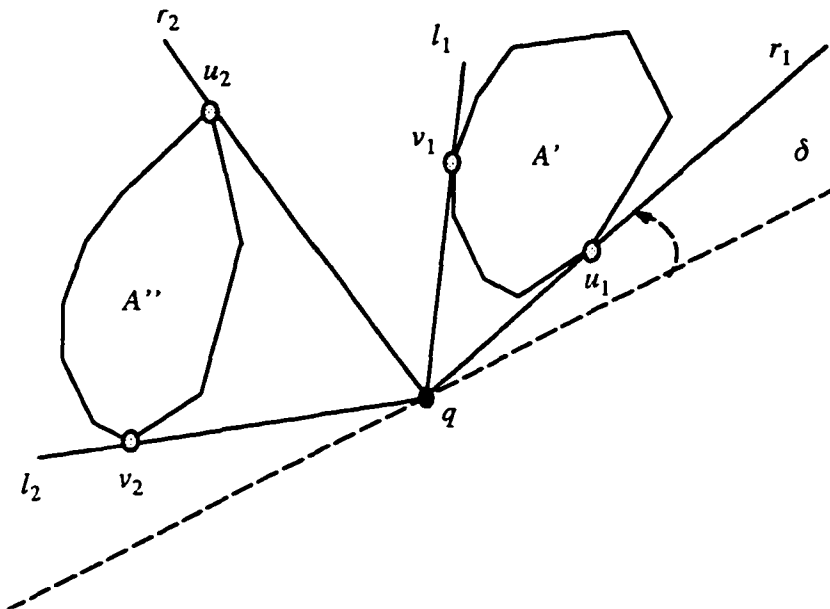


Figure 2.9: *The operation \diamond*

The condition of Lemma 2.5 can be tested very efficiently: the only check that is necessary is to detect whether the ray opposite to one of $\overrightarrow{qu_1}$, $\overrightarrow{qu_2}$, $\overrightarrow{qv_1}$, and $\overrightarrow{qv_2}$ intersects the other wedge. This can be tested in the obvious way in constant time. Moreover, a processor detecting that condition of Lemma 2.5 holds for the

wedges (u_1, q, v_1) and (u_2, q, v_2) will set the result to $(0, q, 2\pi)$ which is the wedge centered at q and encompassing the whole plane.

It is, therefore, assumed that when combining the wedges (u_1, q, v_1) and (u_2, q, v_2) the condition of Lemma 2.5 does not hold. Put differently, there exists an infinite line δ through q such that both the wedges (u_1, q, v_1) and (u_2, q, v_2) lie in one halfplane with respect to δ , as illustrated in Figure 2.9. This motivates definitions $\min\{r_1, r_2\}$ and $\max\{l_1, l_2\}$ as the bounding rays of the union of the two wedges. Specifically, $\min\{r_1, r_2\}$ is the ray encountered first as δ is rotated counter-clockwise about q , while $\max\{l_1, l_2\}$ is the ray encountered last. For example, in Figure 2.9 $\min\{r_1, r_2\} = r_1$ and $\max\{l_1, l_2\} = l_2$. In this terminology, the binary operation \diamond on these wedges is defined as follows:

$$(r_1, q, l_1) \diamond (r_2, q, l_2) = \begin{cases} (\min\{r_1, r_2\}, q, \max\{l_1, l_2\}) & \text{if the condition of Lemma 2.5. does not hold} \\ (0, q, 2\pi) & \text{otherwise.} \end{cases} \quad (2.2)$$

Clearly, \diamond either captures the fact that q lies in the interior of $A' \cup A''$ in which case q will surely lie inside the convex hull of A , or it returns the right and left bounding rays of the wedge centered at q and containing the points in $A' \cup A''$. Furthermore, the operation \diamond in (2.2) is both associative and commutative, and so the computation of Stage 3 will yield the desired result. To summarize the findings the following result is stated.

Theorem 2.6. An arbitrary instance of the MULTI-LOCATION problem involving sets A and Q of cardinalities n and m , respectively, can be solved in $O(T_M(M, \frac{p \cdot M}{m}) + ((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p \cdot M}{m}))$ on an $\text{ACM}(n, p, M)$. \square

The MULTI-LOCATION algorithm can be extended to solve the following related problems:

1. CONTAINMENT: Determine whether the convex hull of Q (resp. A) is con-

- tained in the convex hull of A (resp. Q);
2. SEPARABILITY: Determine whether the sets A and Q are linearly separable and if so, find a separating line;
 3. COMMON-TANGENTS: In case A and Q are separable, find their common supporting lines (i.e., tangents).

The procedure for the CONTAINMENT problem is as follows. First, to detect whether the convex hull of Q lies inside of the convex hull of A , the algorithm for MULTI-LOCATION is used. For each point $q_i \in Q$ associate a bit b_i . The value of b_i is set to 1 if q_i lies outside $\text{CH}(A)$, 0 otherwise. Now the problem at hand is the classic OR problem. If the OR of the bits b_i ($1 \leq i \leq m$) is zero then clearly the convex hull of Q lies inside that of A . If there exist two bits b_i and b_j such that $b_i = 0$ and $b_j = 1$ then the two sets intersect without any containment.

To decide whether A lies within the convex hull of Q the following procedure is used. The $\text{ACM}(n, p, M)$ is partitioned into groups G_i ($1 \leq i \leq \frac{p}{s}$), as in Stage 1 of the MULTI-LOCATION problem. Further, the points in Q are replicated in every group G_i . Next, in every processor belonging to such a group the convex hull of the subset of points in Q is computed using an optimal algorithm [67] and every point of A that lies in G_i checks whether it is interior to the convex hull of Q . It is noted that a similar procedure described in the Stage 2 of the MULTI-LOCATION algorithm will work here. As the same query set Q is present in every G_i , every point in A determines its own status. The solution can now be determined by solving the corresponding instance of the OR problem.

The OR can be computed by a reduce operation. The running times these reduce operations is dominated by other operations of the algorithm. Consequently the following result is stated.

Theorem 2.7. An arbitrary instance of the CONTAINMENT problem involving sets A and Q of cardinalities n and m , respectively, can be solved in $O(T_M(M, \frac{p^*M}{m}) + ((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p^*M}{m}))$ on an $ACM(n, p, M)$. \square

To solve the SEPARABILITY problem the following approach is used. Firstly, solve the MULTI-LOCATION problem. If any of the points in Q is interior to the convex hull of A , then A and Q are not separable. Therefore it is assumed that every point of Q is exterior to $CH(A)$. Recall that the MULTI-LOCATION algorithm provides every point in Q with a “certificate” for being exterior to $CH(A)$: for every point in Q this certificate is a pair of supporting lines to $CH(A)$. For every query point q_i of Q , let l_i and r_i be the left and right supporting lines from q_i to $CH(A)$, respectively.

Next, the following instance of the MQ problem is solved problem with the following parameters:

- the set of items is the set Q ,
- the set of queries is the set of lines $\mathcal{L} = \{l_i, r_i \mid 1 \leq i \leq m\}$,
- for every ordered pair $(d, q) \in \mathcal{L} \times Q$, $\phi(d, q) = \text{“yes”}$ if q lies in the closed halfplane determined by d not containing the interior of A ,
- let S_d be the set of points q in Q for which $\phi(d, q) = \text{“yes”}$; now $f(S_d) = d$ in case $|S_d| = m$ and \emptyset otherwise.

It is clear that, in this formulation, the solution to the corresponding instance of the MQ problem returns the two separating lines for A and Q , thus solving the SEPARABILITY problem. The algorithm for solving the instance of the MQ problem stated above proceeds along lines identical to those of MULTI-LOCATION discussed above and is, therefore, omitted. A similar formulation applies for the

COMMON-TANGENTS problem except that A and Q should be on the same side of the supporting line. Consequently, the following result is obtained.

Theorem 2.8. An arbitrary instance of the SEPARABILITY and COMMON-TANGENTS problems involving sets A and Q of cardinalities n and m , respectively, can be solved in $O(T_M(M, \frac{p \cdot M}{m}) + ((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p \cdot M}{m}))$ on an $ACM(n, p, M)$. \square

2.4 Proximity-Related Computations

The purpose of this section is to show that four fundamental problems in pattern recognition, robotics, and image processing can be solved elegantly by stating them as instances of the MQ problem.

2.4.1 The Multiple Closest Segment Problem

Given a set A of non-intersecting line segments and a set Q of points in the plane, the *multiple closest segment* problem is to determine for each point in Q , the closest segment in A (if any) intersected by vertical rays emanating from it. This problem will be referred to as CLOSEST-SEGMENT. For an illustration, refer to Figure 2.10. It is well known that the CLOSEST-SEGMENT problem finds numerous applications ranging from visibility, to ray tracing, to robotics, to name just a very few [7, 29, 50, 75].

As it turns out, the CLOSEST-SEGMENT problem can be stated as a MQ problem. For definiteness, let $A = \{a_1, a_2, \dots, a_n\}$ and $Q = \{q_1, q_2, \dots, q_m\}$. The corresponding instance of the MQ problem has the following parameters:

- the set $A = \{a_1, a_2, \dots, a_n\}$ of segments,

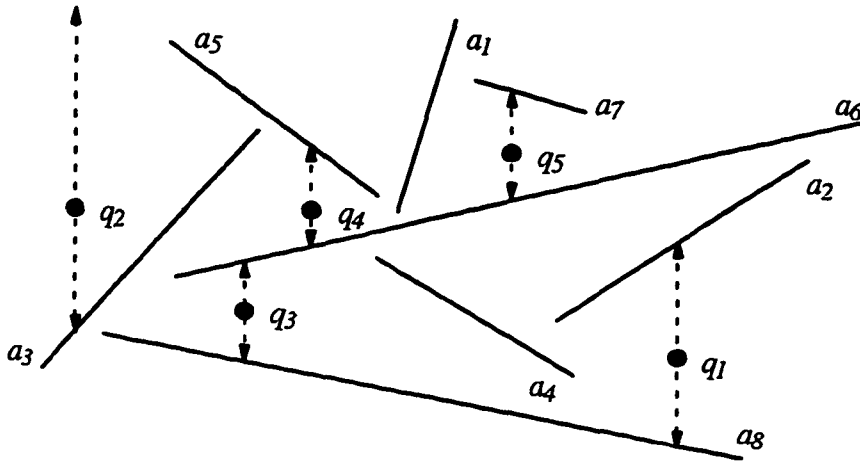


Figure 2.10: *The CLOSEST-SEGMENT problem*

- the set $Q = \{q_1, q_2, \dots, q_m\}$ of query-points,
- a decision problem $\phi : Q \times A \rightarrow \{\text{"yes"}, \text{"no"}\}$ such that $\phi(q_i, a_j) = \text{"yes"}$ whenever a_j is the closest segment above (resp. below) q_i intersected by vertical rays originating at q_i ,
- a function f such that $f(S) = S$.

The compute-and-move operation for CLOSEST-SEGMENT proceeds as follows.

Stage 2. Begin by computing the trapezoidal decomposition (vertically) of the segments in every processor this can be done using the classic trapezoidal decomposition, algorithm [67]. For a more detailed discussion of trapezoidal decomposition refer to [67]. It is noted that the closest segments for a query q can be found by simply locating the trapezoid in which q lies.

Consider two sets of segments S' and S'' . Let l_b', l_t' and l_b'', l_t'' be the closest segments of q in the sets S' and S'' , respectively. The closest segments of q in the set $S' \cup S''$ can be found by picking the closer of the two solutions in S' and S'' . This will ensure that along with each query, as the compute-and-move operations proceed

the closest segments of all the segments encountered so far can be maintained.

Trapezoidal decomposition can be done in $O(\frac{n}{p} \log \frac{n}{p})$ and point location for a single *query* will take $\log \frac{n}{p}$ time. For every query point in Q , the closest segments among the subset of A in each G_i can be found in $O((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p})))$ time. Consequently, the following result is proved.

Theorem 2.9. An arbitrary instance of the CLOSEST-SEGMENT problem involving a set of n non-intersecting line segments and a set of m points in the plane, can be solved in $O(T_M(M, \frac{p \cdot M}{m}) + ((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p \cdot M}{m}))$ on an $ACM(n, p, M)$. \square

2.4.2 The Multiple Circle Problem

Given a set A of points in the plane and a set Q of disjoint circles, the *multiple circle* problem is to determine for each circle the number of points in A it contains. This problem is referred to as the MULTI-CIRCLE. The MULTI-CIRCLE can be seen as a natural generalization of the well-known facility location problem involving a set of existing sites and a collection of proposed facilities (radio stations, for example) to be placed. In this context one is interested in computing, for each of the facilities, the number of points it will service. The MULTI-CIRCLE problem finds numerous applications to geographic data processing, facility location, robot navigation, visibility, among many others [29, 46, 50, 74]. Refer to Figure 2.11 for an instance of the MULTI-CIRCLE problem.

The MULTI-CIRCLE problem can be stated as an instance of the MQ problem in the following way. For definiteness, let $A = \{a_1, a_2, \dots, a_n\}$ and $Q = \{q_1, q_2, \dots, q_m\}$. The corresponding instance of the MQ problem has the following parameters:

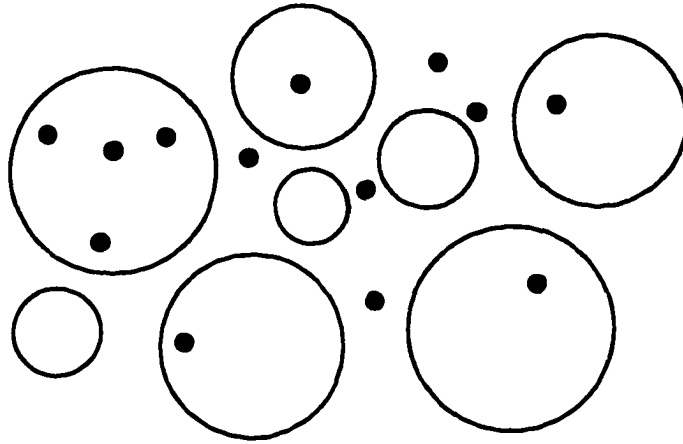


Figure 2.11: *The MULTI-CIRCLE problem*

- the set $A = \{a_1, a_2, \dots, a_n\}$ of points,
- the set $Q = \{q_1, q_2, \dots, q_m\}$ of circles,
- a decision problem $\phi : Q \times A \rightarrow \{\text{"yes"}, \text{"no"}\}$ is such that $\phi(q_i, a_j) = \text{"yes"}$ whenever a_j is inside the circle q_i ,
- $f(S) = |S|$.

The MULTI-CIRCLE problem can be solved by using a similar procedure as the CLOSEST-SEGMENT. This is done as follows: each circle q_i is replaced with its diameter d_i which is parallel to the x -axis. The difference between this problem and the CLOSEST-SEGMENT is that, here segments are queries and for each segment the items which are within the corresponding circle are to be determined. This can be achieved in the following way, just as in the CLOSEST-SEGMENT each item point a can determine the closest segments d_j and d_k from above and below, respectively. The item a lies in the circle q_j (corresponding to d_j), if and only if the distance between a and the center of q_j is less than the radius of q_j , refer to Figure 2.12. The same check is repeated for q_k . Each item can belong to a unique circle

(non-overlapping circles). Once all the items in a processor determine the query circles they belong to, a simple scan will be used to determine the number of points in each query circle. The running time for this processing is same as that of the CLOSEST-SEGMENT. Consequently, the following result is proved.

Theorem 2.10. An arbitrary instance of the MULTI-CIRCLE problem involving a set A of n points in the plane and a set Q of m disjoint circles, can be solved in $O(T_M(M, \frac{p \cdot M}{m}) + ((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p \cdot M}{m}))$ on an $ACM(n, p, M)$. \square

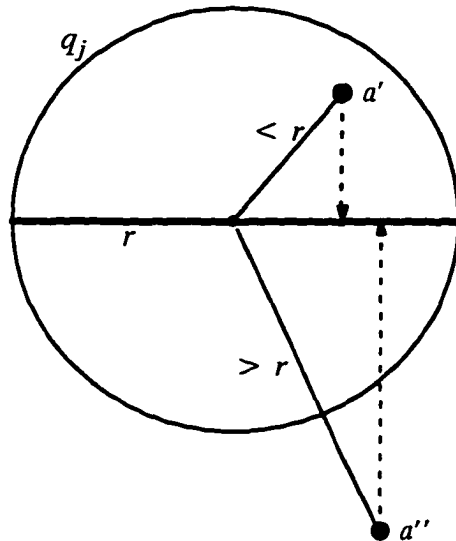


Figure 2.12: *The reduction of MULTI-CIRCLE to CLOSEST-SEGMENT*

2.4.3 The Multiple Range Problem

Given a set A of points in the plane and a set Q of non-overlapping rectangles, the *multiple range* problem is to determine for each rectangle in Q the number of points in A it contains. This problem is referred to as MULTI-RANGE. The MULTI-RANGE

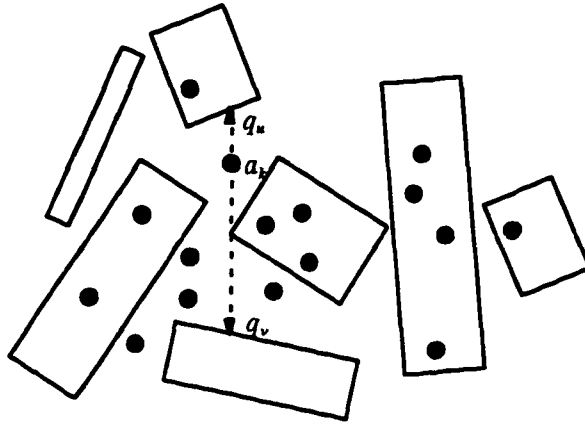


Figure 2.13: *The MULTI-RANGE problem*

problem can be seen as a natural generalization of the well-known range query problem involving a set of points in the plane and one query rectangle. Just like the range query problems, the MULTI-RANGE problem finds numerous applications to geographic data processing, facility location, robot navigation, visibility, ray tracing, VLSI compaction, to name just a very few [7, 29, 35, 46, 50, 74, 75]. Refer to Figure 2.13 for an instance of the MULTI-RANGE problem featuring 16 points and 7 rectangles.

The power of the paradigm is demonstrated again by proving that the MULTI-RANGE can be formulated as an instance of the MQ problem. For definiteness, let $A = \{a_1, a_2, \dots, a_n\}$ and $Q = \{q_1, q_2, \dots, q_m\}$. The corresponding instance of the MQ problem has the following parameters:

- the set $A = \{a_1, a_2, \dots, a_n\}$ of points,
- the set $Q = \{q_1, q_2, \dots, q_m\}$ of rectangles,
- a decision problem $\phi : Q \times A \rightarrow \{\text{"yes"}, \text{"no"}\}$ is such that $\phi(q_i, a_j) = \text{"yes"}$ whenever a_j is inside rectangle q_i ,

- $f(S) = |S|$.

This algorithm is similar to that of MULTI-CIRCLE. The details of compute-and-move operation follow.

The collection of rectangles is viewed as consisting of $4m$ line segments[†]. Just as in the Stage 2 of the MULTI-CIRCLE, for each item point a_k determine the identity of the closest line segments (i.e., rectangles) met by vertical rays emanating from it. It is easy to confirm that a point a_k is inside a rectangle q_t if and only if both the segments obtained belong to the rectangle q_t . Note that the four segments of a rectangle belong to the same processor. Referring again to Figure 2.13, observe that point a_k lies in none of the rectangles in the collection: this is confirmed by the fact that the closest segments intersected by vertical rays emanating from a_k belong to different rectangles q_u and q_v .

The running time is the same as that of MULTI-CIRCLE. Consequently the following result is obtained.

Theorem 2.11. An arbitrary instance of the MULTI-RANGE problem involving a set A of n points in the plane and a set Q of m non-overlapping rectangles, can be solved in $O(T_M(M, \frac{p \cdot M}{m}) + ((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p \cdot M}{m}))$ on an ACM(n, p, M). \square

2.4.4 The Multiple Closest Point Problem

For two points p and q let $d(p, q)$ stand for the Euclidian distance between them. Given sets A and Q of points in the plane, the *multiple closest point* problem is to determine for each point in Q , a point in A that is closest to it in the Euclidian distance sense. This problem is referred to as CLOSEST-POINT, it is a fundamental

[†]Note that the four segments of a rectangle belong to the same processor.

problem that finds additional applications in geographic data processing, computer graphics, image processing, and morphology, to name just a few [1, 75].

The formulation of CLOSEST-POINT as an instance of the MQ problem. For definiteness, let $A = \{a_1, a_2, \dots, a_n\}$ and $Q = \{q_1, q_2, \dots, q_m\}$, further define the parameters as follows:

- the set $A = \{a_1, a_2, \dots, a_n\}$ of points,
- the set $Q = \{q_1, q_2, \dots, q_m\}$ of query-points,
- a decision problem $\phi : Q \times A \rightarrow \{\text{"yes"}, \text{"no"}\}$ is such that $\phi(q_i, a_j) = \text{"yes"}$ whenever $d(q_i, a_j) = \min_{1 \leq k \leq n} d(q_i, a_k)$,
- for every i ($1 \leq i \leq m$), let $S_i = \{a_j \in A \mid \phi(q_i, a_j) = \text{"yes"}\}$; $f(S_i) = \min\{j \mid a_j \in S_i\}$, in other words, the solution to query q_i is the point with the smallest subscript that is closest to q_i .

The compute-and-move operation for CLOSEST-POINT is as follows.

The processing will be partitioned into two substages. In the first substage, the Voronoi diagram of the subset of points of A located in every processor is constructed. This task can be performed in $O(\frac{n}{p} \log \frac{n}{p})$ time using the optimal sequential algorithm described in [67]. Note that in the move part of the compute-and-move operation, this voronoi diagram is passed on to the next processor instead of the items.

In the second substage, for every point in Q the Voronoi polygon that contains it is determined. Once the identity of the enclosing Voronoi polygon is known, the local instance of the CLOSEST-POINT problem is, essentially, solved. The problem at hand can be solved efficiently by observing that the total number of edges of the Voronoi diagram of the subset of point of A located in P_i is in $O(\frac{n}{p})$.

A further key observation is that to identify, for every point q in Q the unique enclosing Voronoi polygon, it is sufficient to identify the first Voronoi edge intersected by a ray originating at q and going in the positive y -direction. This is again an instance of the CLOSEST-SEGMENT problem. Consequently, the following result is obtained.

Theorem 2.12. An arbitrary instance of the CLOSEST-POINT problem involving sets A and Q of size n , and m , respectively, can be solved in $O(T_M(M, \frac{p \cdot M}{m}) + ((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p \cdot M}{m}))$ on an $\text{ACM}(n, p, M)$. \square

2.5 Stabbing-Related Problems

Let $A = \{a_1, a_2, \dots, a_n\}$ be an arbitrary set of possibly intersecting line segments in the plane and let $Q = \{q_1, q_2, \dots, q_m\}$, ($1 \leq m \leq n$), be a set of parallel lines. The lines in Q will be referred to as *query* lines. The *multiple stabbing* problem, (MULTI-STABBING, for short) asks to determine for every query line q_i , the number of segments in A it intersects. Figure 2.14 features an instance of the MULTI-STABBING involving a set of four query lines. The MULTI-STABBING problem is a natural generalization of the stabbing line problem [1] that involves only one such query-line. The stabbing line problem finds applications to computer graphics, path planning [50], and morphology [75]. The purpose of this section is to show an elegant solution to the MULTI-STABBING problem by reducing it to an instance of the MQ problem.

Without loss of generality, it is assumed that all the query lines are parallel to the x -axis and that the line segments are in general position, with no two endpoints sharing the same y -coordinate. Every line segment a_i is specified by its top and bottom endpoints, t_i and b_i , respectively.

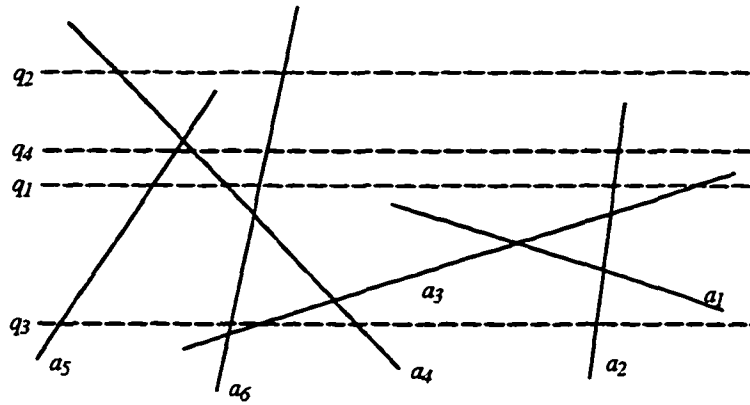


Figure 2.14: An instance of the MULTI-STABBING problem

The MULTI-STABBING problem can be stated as an instance of the MQ problem with the following parameters:

- the set $A = \{a_1, a_2, \dots, a_n\}$ of items is precisely the given set of line segments,
- the set $Q = \{q_1, q_2, \dots, q_m\}$ of queries consists of the m query-lines,
- a decision problem $\phi : Q \times A \rightarrow \{\text{"yes"}, \text{"no"}\}$ such that $\phi(q_i, a_j) = \text{"yes"}$ if and only if the query-line q_i intersects segment a_j ,
- $f(S) = |S|$.

For every q_i ($1 \leq i \leq m$), let $S_i = \{a_j \in A \mid \phi(q_i, a_j) = \text{"yes"}\}$. $f(S_i)$ is the number of line segments "stabbed" by query-line q_i .

The compute-and-move operation for MULTI-STABBING is detailed as follows.

Consider the subset of line segments of A in P_i as A_i . Begin by sorting the subset A_i of line segments in each P_i in decreasing order of the y -coordinate of their top and bottom endpoints. The sorting can be performed in $O(\frac{n}{p} \log \frac{n}{p})$ time using any optimal sorting algorithm. Let $e_1, e_2, \dots, e_{2\frac{n}{p}}$ be the resulting sequence of endpoints. The processing in this stage is motivated by the following simple observation whose

proof is immediate.

Observation. Let q_u be a query-line specified by its equation $q_u = y_u$. The number of line segments in P_i stabbed by q_u is precisely the number of line segments whose top endpoint has a higher y -coordinate than y_u and whose bottom endpoint has a lower y -coordinate than y_u . \square

Consider the sorted sequence $e_1, e_2, \dots, e_{2\frac{n}{p}}$ and assign each top endpoint in this sequence a weight of $+1$ and to each bottom endpoint a weight of -1 . Perform a prefix sum on the resulting weighted sequence this takes $O(\frac{n}{p})$ time. It is easy to confirm that for every endpoint e of a line segment in P_i the resulting value of the prefix sum is exactly the number of segments intersected by a horizontal line through e .

Next, identify for every query q_u the unique pair (e_p, e_{p+1}) of endpoints with the property that $e_p > y_u > e_{p+1}$. Once this is done, the desired solution of q_u is the value of the previous prefix sum for e_p . The task of identifying the pair (e_p, e_{p+1}) can be carried out by a simple binary search. Note that the process of sorting and computing the prefix sums for the subset of items need not be done at every compute-and-move operation. These tasks are performed as initialization steps and only the sorted sequence and the prefix sums are communicated. In summary the following result is stated.

Lemma 2.13. The task of computing for every query-line in Q the number of line segments in each group G_i it intersects can be carried out in $O((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p})))$ time. \square

Consequently, the following result is proved .

Theorem 2.14. An arbitrary instance of the MULTI-STABBING problem involving a set of n line segments and a set of m query-lines can be solved in $O(T_M(M, \frac{p \cdot M}{m}) +$

$((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p \cdot M}{m})$ on an $\text{ACM}(n, p, M)$. \square

Let $A = \{a_1, a_2, \dots, a_n\}$ be a simple polygon in the plane and let $Q = \{q_1, q_2, \dots, q_m\}$, $(1 \leq m \leq n)$, be an arbitrary set of points. The *simple polygon location* problem, (POLY-LOCATION, for short) is to determine for every query point q_i whether or not it lies in the interior of A . An instance of the POLY-LOCATION problem is illustrated in Figure 2.15. The POLY-LOCATION is a variant of a large class of point location problems, with applications to computer graphics, facility location, path planning, among others [29, 35, 50].

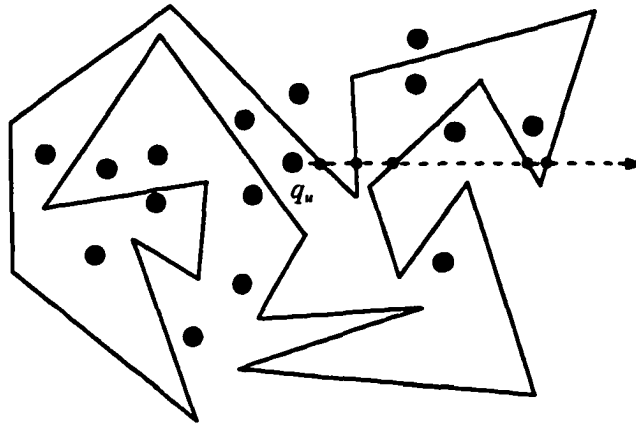


Figure 2.15: *An instance of the POLY-LOCATION problem*

A solution to the POLY-LOCATION problem can be obtained by reducing it to an instance of the MULTI-STABBING problem. This is done as follows. The simple polygon is perceived as a collection of line segments (its edges) and the resulting instance of the MULTI-STABBING problem is solved. However, the counting of intersections is slightly changed. Consider an arbitrary point q_u in Q and let λ_u be the horizontal line through q_u . For q_u , only the number of intersection points of

A and λ_u that lie to the right of q_u , are of interest. Now the Jordan Curve Theorem guarantees that q_u is inside A if and only if the number of intersections recorded is odd. To summarize the findings the following result is stated.

Theorem 2.15. An arbitrary instance of the POLY-LOCATION problem involving an n -vertex simple polygon a set of m query-points can be solved in $O(T_M(M, \frac{p \cdot M}{m}) + ((m + \frac{n}{p}) \log \frac{n}{p} + (\frac{m}{M} - 1)(m \log \frac{n}{p} + T_P(\frac{n}{p}))) + T_R(M, \frac{p \cdot M}{m}))$ on an $ACM(n, p, M)$. \square

CHAPTER 3

THE SORTED MATRIX ALGORITHM ON THE ACM

In the previous chapter, the power of computational paradigm was demonstrated by dealing with some instances of the MQ problem. The main goal of this chapter is to discuss query processing in a structured application domain. Query processing is a crucial transaction in various applications including information retrieval, database design and management, and VLSI. Many of these applications involve data stored in a matrix satisfying a number of properties. One property that occurs time and again specifies that the rows and the columns of the matrix are independently sorted [25, 40, 58, 78]. It is customary to refer to such a matrix as *sorted*. A matrix is said to be *fully sorted* if its entries are sorted in row-major (or column-major) order. Figure 3.1a displays a sorted matrix; Figure 3.1b features a fully sorted version of the matrix in Figure 3.1a.

Sorted matrices provide a natural generalization of a number of real-life situations. Consider vectors $X = (x_1, x_2, \dots, x_{\sqrt{n}})$ and $Y = (y_1, y_2, \dots, y_{\sqrt{n}})$ with $x_i \leq x_j$ and $y_i \leq y_j$, whenever $i < j$. The Cartesian sum of X and Y , denoted $X + Y$ is the $\sqrt{n} \times \sqrt{n}$ matrix A with entries $a_{ij} = x_i + y_j$. It is clear that $X + Y$ is a sorted matrix. Moreover, $X + Y$ can be stored succinctly in $O(\sqrt{n})$ space [25, 32],

1	6	10	19
3	9	14	26
5	15	20	40
7	17	24	41

a

1	3	5	6
7	9	10	14
15	17	19	20
24	26	40	41

b

Figure 3.1: *Sorted and fully sorted matrices*

since the entries a_{ij} can be computed, as needed, in constant time. Searching, ranking, and selection in sorted matrices are key ingredients in fast algorithms in VLSI design, optimization, statistics, database design, and facility location problems and have received considerable attention in the literature [25, 26, 32, 37, 40, 58, 78].

This chapter addresses the problems of batched searching and ranking in sorted matrices. It will be shown that these problems can be formulated as instances of the MQ problem. Consider a sorted matrix A of size $\sqrt{n} \times \sqrt{n}$ of items from a totally ordered universe, $\frac{n}{p}$ items per processor, on an $ACM(n, p, M)$. Also given an arbitrary sequence $Q = q_1, q_2, \dots, q_m$, ($1 \leq m \leq n$), of queries stored M per processor in the first $\frac{m}{M}$ processors of the platform. The queries are of two types: for a query q_j of the first type one is interested in an item of A that is closest to q_j ; for a query q_j of the second type one is interested in the number of items in A that are strictly smaller than q_j . The two query types are referred to as *search queries* and *rank queries*, respectively. The set Q of queries is an arbitrary mix of the two query types. In this context, the Batched Searching and Ranking problem, (BSR, for short) involves determining the solution of every query in Q .

Formulate the search queries as follows:

- the set $A = \{a_1, a_2, \dots, a_n\}$ of items is made up of the elements of the given

sorted matrix,

- the set $Q = \{q_1, q_2, \dots, q_m\}$ of queries,
- the decision problem $\phi : Q \times A \rightarrow \{\text{"yes"}, \text{"no"}\}$ is such that $\phi(q_i, a_j) = \text{"yes"}$,
- $f(S_i) = \min(|q_i - a_j|), 1 \leq j \leq n$.

The formulation for the rank queries is obvious.

It is important to note that search queries occur frequently in image processing, pattern recognition, computational learning, and artificial intelligence, where one is interested in returning the item in the database that best matches, in some sense, the query at hand [7, 29, 74, 82]. On the other hand, rank queries are central to relational database design, histogramming, and pattern analysis [7, 29, 53, 82]. Here, given a collection of items in a database along with a query, one is interested in computing the number of items in the database that have a lesser value than the query [53]. In addition, rank queries finds applications to image processing, robotics, and pattern recognition [7, 11, 29, 46]. It is noted that a variant of rank queries has also received attention in the literature. Specifically, a *range query* involves determining the number of items in a given database that fall in a certain range. It is not hard to see that range queries can be answered by specifying them as rank queries [29].

Throughout this chapter for simplicity of exposition it is assumed that all the queries fit into one processor (i.e., $m \leq M$). This is not a serious restriction as the algorithm can be easily extended to the case $M < m \leq n$. With this assumption in mind, a generic instance of the BSR problem involves a sorted matrix A of size $\sqrt{n} \times \sqrt{n}$ stored $\frac{n}{p}$ items per processor in an $\text{ACM}(n, p, M)$ and a collection Q of m , ($1 \leq m \leq M$), queries stored in processor P_0 of the platform. Moreover, to avoid

handling double subscripts, the items of matrix A will be enumerated, in row major order, as a_1, a_2, \dots, a_n .

The remainder of the chapter discusses the algorithm for the BSR problem.

3.1 BSR Algorithm on the ACM

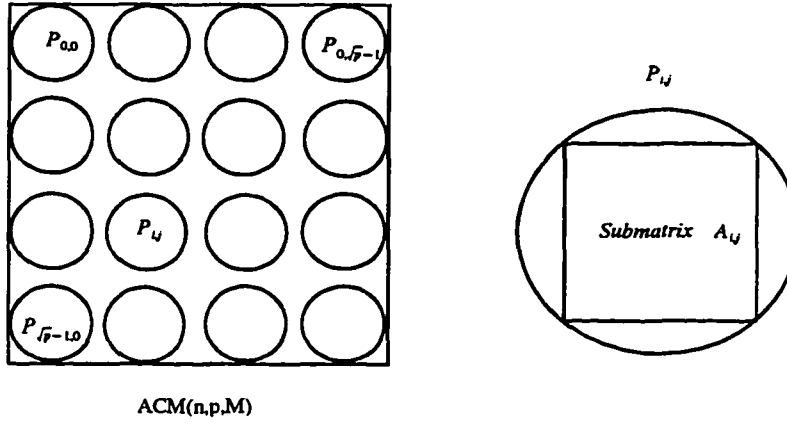


Figure 3.2: *The matrix view of the $ACM(n, p, M)$*

Let the processors of the $ACM(n, p, M)$ be P_0, P_1, \dots, P_p . As the input is a matrix, it will be convenient to view the processors of the $ACM(n, p, M)$ as a matrix of size[†] $\sqrt{p} \times \sqrt{p}$, with processor P_i being the same as $P_{\lfloor \frac{i}{\sqrt{p}} \rfloor, i \bmod \sqrt{p}}$. Superimposing the matrix of processors on the given matrix A naturally defines a block partition of A with processors $P_{i,j}$ storing $A_{i,j}$, refer to Figure 3.2. Specifically, assume that the matrix A is partitioned into p submatrices each of size $\sqrt{\frac{n}{p}} \times \sqrt{\frac{n}{p}}$, denote the $(i, j)^{th}$ submatrix as $A_{i,j}$, refer to Figure 3.3. The sequence of processors belonging to row i (i.e., $P_{i,0}, P_{i,1}, \dots, P_{i,\sqrt{p}-1}$) will be referred to as a *horizontal slice* of $ACM(n, p, M)$ and denoted by HS_i . A *vertical slice*, VS_i , is defined in a dual manner.

[†]For convenience, \sqrt{p} is assumed to be an integer

The algorithm for the BSR problem proceeds in the same lines as the generic algorithm of Chapter 2, that is, the algorithm proceeds in three stages.

Stage 1. The set Q of queries is replicated in each processor $P_{i,j}$, creating local instances of the BSR problem.

Stage 2. Determine in each processor $P_{i,j}$, in parallel, the solution of the local instance of the BSR problem.

Stage 3. The solutions of the local instances of the BSR problem obtained in Stage 2 are combined into the solution of the original BSR problem.

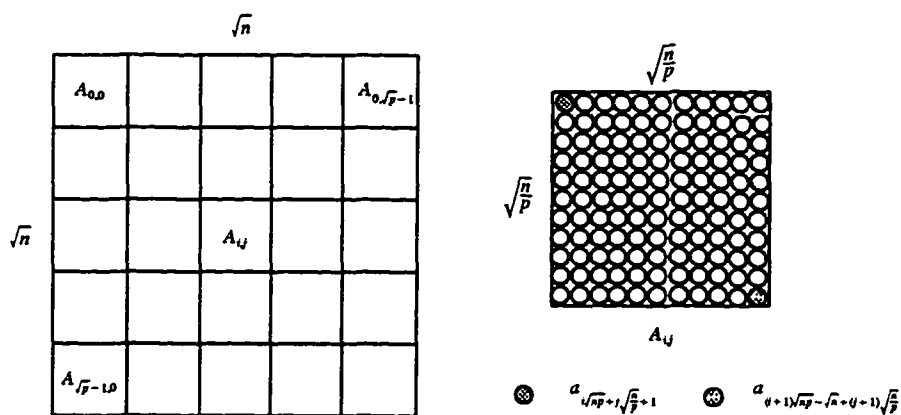


Figure 3.3: *The partition of matrix A*

The remainder of this section is devoted to a detailed description of each of these stages.

Stage 1.

The purpose of this stage is to replicate the set Q of queries, in each processor $P_{i,j}$, m queries per processor. This is a simple broadcast operation, and so its running time will be of the order of $T_B(m, p)$.

Stage 2.

Each local instance involves, $A_{i,j}$, the subset of A stored by the processors in $P_{i,j}$ and the entire set Q of queries.

The main goal of this stage is to solve the local instance of BSR in each processor $P_{i,j}$. Begin by sorting the items and queries in each $P_{i,j}$ using an optimal sequential sorting algorithm. In the sorting process, ties are broken in favor of queries. In other words, if a query and an item are equal, then in the sorted version the query precedes the item.

Let $C_{i,j} = c_1, c_2, \dots, c_{m+\frac{n}{p}-1}, c_{m+\frac{n}{p}}$ be the resulting sorted sequence stored in processor $P_{i,j}$. The following two results will justify the approach to solving the local instances of the BSR problem.

Lemma 3.1. Let q_k be a query of rank type and assume that $c_t = q_k$, in other words, q_k occurs in position t in the sorted sequence $C_{i,j}$. The number of items in $P_{i,j}$ strictly smaller than q_k equals the number of items preceding q_k in $C_{i,j}$.

Proof. Follows directly from the sortedness of $C_{i,j}$ along with the assumed tie-breaking discipline. \square

Lemma 3.1 motivates the following strategy for solving all rank type queries in $P_{i,j}$. Assign to every c_t a weight w_t defined as follows:

$$w_t = \begin{cases} 1 & \text{if } c_t \text{ is an item} \\ 0 & \text{if } c_t \text{ is a query.} \end{cases} \quad (3.1)$$

Next, compute the prefix sums of the sequence $c_1, c_2, \dots, c_{m+\frac{n}{p}-1}, c_{m+\frac{n}{p}}$ using the

weights assigned in (3.1) and let $e_1, e_2, \dots, e_{m+\frac{n}{p}-1}, e_{m+\frac{n}{p}}$ be the result. By virtue of Lemma 3.1, the value e_t corresponding to $c_t = q_k$ is exactly the number of items in $P_{i,j}$ strictly smaller than q_k . The time taken for all the rank queries will be dominated by sorting and prefix sum computations, which is $O((m + \frac{n}{p}) \log(m + \frac{n}{p}))$.

The task of handling search queries requires a different approach. To motivate this strategy, consider again the sorted sequence $C_{i,j} = c_1, c_2, \dots, c_{m+\frac{n}{p}-1}, c_{m+\frac{n}{p}}$ and refer to Figure 3.4.

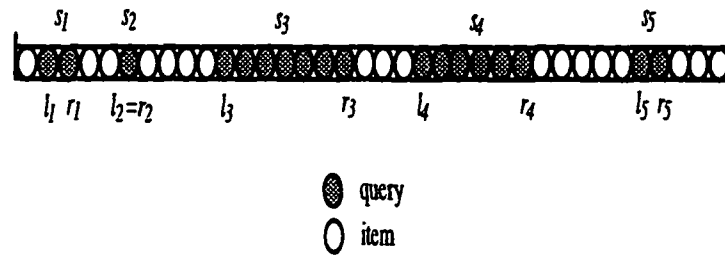


Figure 3.4: The sorted sequence $C_{i,j}$

The m queries occur in $C_{i,j}$ in *contiguous* subsequences s_1, s_2, \dots, s_d ; for every such sequence s_p let l_p and r_p stand, respectively, for the leftmost and rightmost query in s_p , as illustrated in Figure 3.4. Of course, if the sequence s_p consists of one query only then $l_p = r_p$. Write $l_p = c_\alpha$ and $r_p = c_\beta$ for some α and β satisfying $1 \leq \alpha \leq \beta \leq m + \frac{n}{p}$. This terminology becomes clear from the following observation.

Lemma 3.2. For all the search queries in some sequence s_p the solution is either $c_{\alpha-1}$ or $c_{\beta+1}$.

Proof. Let q_k be an arbitrary search query in the sequence s_p . The sortedness of $C_{i,j}$ along with the tie-breaking discipline guarantee that no item in $P_{i,j}$ is closer to q_k than one of the items $c_{\alpha-1}$ or $c_{\beta+1}$. \square

In turn, Lemma 3.2 suggests the following approach to solving all the search

queries in $P_{i,j}$. First, assign to every c_t a weight w_t defined as follows:

$$w_t = \begin{cases} c_t & \text{if } c_t \text{ is an item} \\ -\infty & \text{if } c_t \text{ is a query.} \end{cases} \quad (3.2)$$

Next, compute the prefix maxima of the sequence $c_1, c_2, \dots, c_{m+\frac{n}{p}-1}, c_{m+\frac{n}{p}}$ using the weights assigned in (3.2) and let $e_1, e_2, \dots, e_{m+\frac{n}{p}-1}, e_{m+\frac{n}{p}}$ be the result. It is easy to confirm that for every search query $c_t = q_k$, the corresponding value e_t is exactly the identity of the item $c_{\alpha-1}$ (from the previous terminology), or $-\infty$ if no such item exists.

Now, assign to every c_t a weight w_t

$$w_t = \begin{cases} c_t & \text{if } c_t \text{ is an item} \\ +\infty & \text{if } c_t \text{ is a query,} \end{cases} \quad (3.3)$$

and compute the prefix minima of the sequence $c_{m+\frac{n}{p}}, c_{m+\frac{n}{p}-1}, \dots, c_2, c_1$ using the weights assigned in (3.3). Let $e_1, e_2, \dots, e_{m+\frac{n}{p}-1}, e_{m+\frac{n}{p}}$ be the result. It is easy to confirm that for every search query $c_t = q_k$, the corresponding value e_t is exactly the identity of the item $c_{\beta+1}$ (from the previous terminology), or $+\infty$ if no such item exists. Therefore, at the end of these two computations, every search query q_k becomes aware of $c_{\alpha-1}$ or $c_{\beta+1}$. By virtue of Lemma 3.2, this is sufficient for the purpose of determining the solution of every search query q_k in $P_{i,j}$. To summarize the findings the following result is stated.

Lemma 3.3. The task of solving the local instance of the BSR problem in each processor $P_{i,j}$ can be performed, in parallel, in $O((m + \frac{n}{p}) \log(m + \frac{n}{p}))$ time. \square

Stage 3.

At the end of Stage 2, each processor $P_{i,j}$ stores its local solution $\sigma(i, j, k)$ along with query q_k . In case q_k is a search query $\sigma(i, j, k)$ denotes the item in A closest to q_k ; in case q_k is a rank query $\sigma(i, j, k)$ denotes the number of items in

A that are strictly smaller than q_k . The goal of Stage 3 is to combine these local solutions into the solution of q_k in the original instance of the BSR problem.

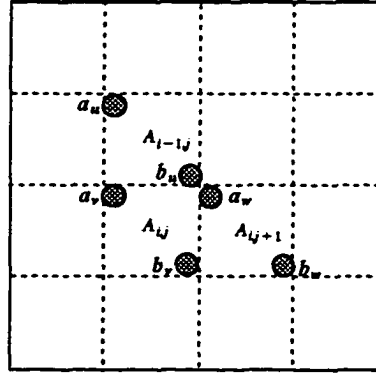


Figure 3.5: *Illustrating the proof of Lemma 3.4.*

In preparation for this, the first task of this stage is to arrange, in every processor $P_{i,j}$, the ordered pairs $(q_k, \sigma(i, j, k))$ sorted by subscript k .

From now on, the processing relies heavily on a technical property of sorted matrices that is discussed next. Referring to Figure 3.5, a processor $P_{i,j}$ is said to be *critical* with respect to a query q_k if q_k is larger than the entry a_v in the northwest corner of $A_{i,j}$ but not greater than the entry b_v in the southeast corner of $A_{i,j}$, in other words:

$$a_v < q_k \leq b_v. \quad (3.4)$$

The following result is key in deriving a time-optimal algorithm for the BSR problem.

Lemma 3.4. If a processor $P_{i,j}$ is critical with respect to a query q_k , then at most one of the processors $P_{i-1,j}$ and $P_{i,j+1}$ may be critical with respect to q_k .

Proof. Referring, again, to Figure 3.5, let a_u, a_v , and a_w stand for the items in the northwest corner of $A_{i-1,j}$, $A_{i,j}$, and $A_{i,j+1}$, respectively. Similarly, let b_u, b_v , and b_w

stand for the items in the southeast corner of $A_{i-1,j}$, $A_{i,j}$, and $A_{i,j+1}$, respectively.

Assume, further, that $P_{i,j}$ is critical with respect to query q_k . Now, if $P_{i-1,j}$ is critical with respect to q_k , then it implies that

$$a_u < q_k \leq b_u \quad (3.5)$$

and, since the matrix A is sorted

$$b_u \leq a_w \leq b_w. \quad (3.6)$$

Now (3.5) and (3.6) combined guarantee that

$$q_k \leq a_w$$

and so, by (3.4), $P_{i,j+1}$ cannot be critical with respect to q_k .

Similarly, if $P_{i,j+1}$ is critical with respect to q_k , then

$$a_w < q_k \leq b_w \quad (3.7)$$

and, since the matrix A is sorted

$$a_u \leq b_u \leq a_w. \quad (3.8)$$

Now (3.7) and (3.8) combined guarantee that

$$b_u < q_k,$$

confirming, by virtue of (3.4), that $P_{i-1,j}$ cannot be critical with respect to q_k . This completes the proof of Lemma 3.4. \square

Consider a generic horizontal slice HS_i . For further reference, a copy of query q_k in some processor $P_{i,j}$ is termed *active* if one of the conditions (a1)–(a4) below is satisfied, refer to Figure 3.6 for an illustration.

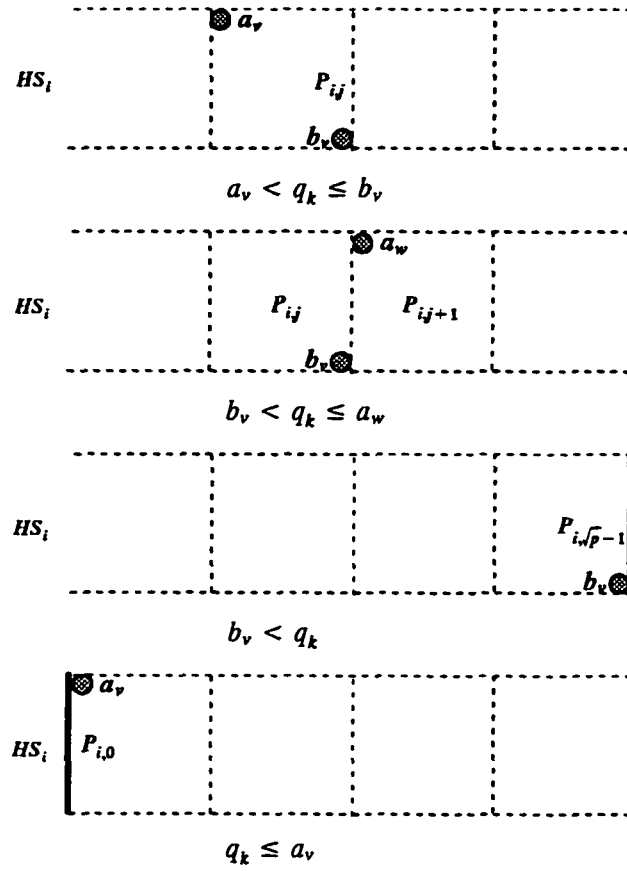


Figure 3.6: *The concept of active copy of query q_k*

- (a1) $P_{i,j}$ is critical with respect to query q_k .
- (a2) Slice HS_i contains no critical processor with respect to query q_k and, for some $j < \sqrt{p} - 1$, q_k is larger than all items in $P_{i,j}$ but smaller than or equal to all items in $P_{i,j+1}$.
- (a3) Query q_k is larger than all the items in slice HS_i ; in this case the copy of q_k in $P_{i,\sqrt{p}-1}$ is active.
- (a4) Query q_k is smaller than or equal to all the items in slice HS_i ; in this case the copy of q_k in $P_{i,0}$ is active.

The leftmost processor of a horizontal slice containing an active copy of a query q_k will be referred to as *leading* with respect to q_k . At this point, one may wonder how all this information is computed. It is clear that for determining what processors $P_{i,j}$ are critical with respect to a given query, only the values in the northwest and southeast corners of the submatrix $A_{i,j}$ are sufficient. Next, every processor $P_{i,j}$ has to be informed about the values of the items in the northwest and southeast corners of the neighboring processors in its own slice. This information can be obtained initially as a preprocessing step. With this information available, critical processors and active copies of all queries can be found in time $O(m)$ time.

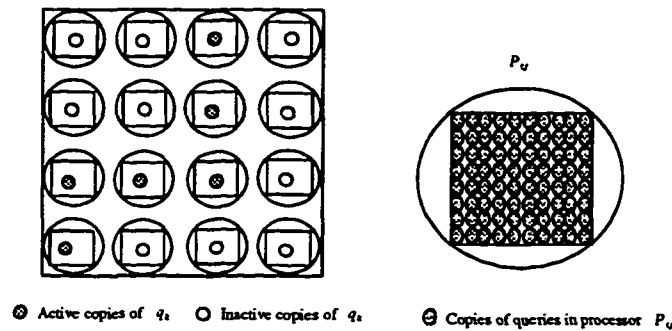


Figure 3.7: *The active copies of query q_k*

The strategy for combining the solutions of queries in every $P_{i,j}$ into the global solution involves gathering of the local solutions horizontally and vertically. This is motivated by Lemma 3.4 and the following observation.

Observation. Consider two adjacent slices HS_i and HS_{i+1} . Let processors $P_{i,j}$ and $P_{i+1,k}$ be *leading* processors with respect to query q . Then in slice HS_i , active copies of q can lie only in processors $P_{i,j}, P_{i,j+1}, \dots, P_{i,k}$. Also for $j < l < k$, $P_{i,l}$ is the only processor which can contain an active query in the vertical slice VS_l with

respect to q (refer to Figure 3.7).

Proof. This follows directly from lemma 3.4. \square

This observation suggests the following rules for gathering of the *active* copies of queries (as illustrated in Figure 3.8).

- (r1) The copy of q_k that belongs to the leading processor in slice HS_i will be scheduled to be gathered in processor $P_{i,0}$.
- (r2) All the remaining active copies of q_k in HS_i will be gathered in the first processor of their vertical slice.

The following result shows that rules (r1) and (r2) lead to one active copy of a query per gather operation.

Lemma 3.5. In a generic slice, at most one active copy per query (say q_k) will be involved in the corresponding gather operation.

Proof. To begin, consider horizontal slices. When a copy of q_k is involved in a horizontal gather operation, then either there exists only one active copy of q_k in slice HS_i (in case the copy of q_k in the leading processor is active by rules (a2)–(a4)) and no other copy of q_k will be in a gather operation in this slice, or else, the copy comes from a leading processor. By rule (r2), all the other active copies in the same slice will be in a vertical gather operation.

Next, consider vertical slices. Suppose that more than one copy of a query is involved in a vertical gather operation, let i be the largest subscript for which the copy of q_k in slice HS_i has more than one copy involved in a vertical gather operation. Without loss of generality, assume that q_k belongs to $P_{i,j+1}$. The conclusion of Lemma 3.4, along with the maximality of i imply that the copy of q_k in processor $P_{i-1,j+1}$ is also using the same vertical bus. This implies that neither $P_{i-1,j+1}$ nor $P_{i,j+1}$ are leading processors (with respect to q_k) in slice HS_{i-1} , and

HS_i , respectively. However, now $P_{i-1,j}$, $P_{i,j}$, and $P_{i,j+1}$ must be critical with respect to q_k , contradicting Lemma 3.4. \square

It is important to note that the total number of active copies of any query q_k is at most $2\sqrt{p}$. This follows immediately from Lemma 3.5, since in each horizontal (vertical) gather operation only one active copy of a query can participate and there are at most \sqrt{p} horizontal (\sqrt{p} vertical) gather operations in all, the conclusion follows.

Next, one may wonder if the active copies of query q_k carry enough information to yield the correct overall solution of q_k . The answer to this natural question is provided by the following results.

Lemma 3.6. Let q_k be a search query and let a be an item in A closest to q_k . There exists an active copy of q_k in some processor $P_{i,j}$ such that either $a = \sigma(i, j, k)$ or $a = \sigma(i, j - 1, k)$ or $a = \sigma(i, j + 1, k)$.

Proof. By assumption, a must be the solution $\sigma(p, q, k)$ of q_k in some processor $P_{p,q}$. In fact, since the items in the matrix are not necessarily distinct, it is possible that a is the solution of q_k in a number of such processors. Assume, without loss of generality, that such is the case for some processors in slice HS_i . Specifically, let $P_{i,j}$ be the *leftmost* processor in HS_i for which $a = \sigma(i, j, k)$. If the copy of q_k in $P_{i,j}$ is active, there is nothing to prove. Therefore, the copy of q_k in $P_{i,j}$ is assumed to be inactive.

Now, to prove that at least one of the copies of q_k in $P_{i,j-1}$ or $P_{i,j+1}$ is active. Since the copy of q_k in $P_{i,j}$ is inactive, (3.4) guarantees that $P_{i,j}$ cannot be critical with respect to q_k . Therefore, with a_v and b_v denoting, respectively, the item in the northwest and southeast corner of $A_{i,j}$,

$$q_k \leq a_v \text{ or } q_k > b_v. \quad (3.9)$$

Notice that $a = \sigma(i, j, k)$ along with (3.9) implies that a must be either a_v or b_v . Symmetry and without loss of generality, allows for the assumption that $a = a_v$. In turn, this implies

$$q_k \leq a_v. \quad (3.10)$$

Notice that (3.10) along with the fact that the copy of q_k in $P_{i,j}$ is inactive guarantees, by virtue of (a4) that $j \neq 1$ and, thus, $P_{i,j-1}$ must exist. Let, a_u and b_u be the items in the northwest and southeast corner of $A_{i,j-1}$, respectively. Since $P_{i,j}$ is the leftmost processor in HS_i for which $a = \sigma(i, j, k)$ and since the matrix A is sorted, it implies that

$$a_u < q_k. \quad (3.11)$$

Moreover, it is not possible to have $q_k > b_u$ for otherwise, (a2) and (3.11) combined would guarantee that the copy of q_k in $P_{i,j}$ must be active. Therefore, it must be the case that

$$q_k \leq b_u. \quad (3.12)$$

However, equations (3.4), (3.11), and (3.12), combined imply that the copy of q_k in $P_{i,j-1}$ must be active, as desired. This completes the proof of Lemma 3.6. \square

Lemma 3.6 suggests an obvious way of updating the solutions of active copies of a search query q_k . The details are spelled out in the following.

- If the active copy of q_k belongs to a critical processor $P_{i,j}$ and $P_{i,j-1}$ is not critical, then the copy of q_k in $P_{i,j}$ updates its solution $\sigma(i, j, k)$ by combining it with $\sigma(i, j-1, k)$.
- If the active copy of q_k belongs to a critical processor $P_{i,j}$ and $P_{i,j+1}$ is not critical, then the copy of q_k in $P_{i,j}$ updates its solution $\sigma(i, j, k)$ by combining it with $\sigma(i, j+1, k)$.

- If the copy of q_k is active because of rule (a2), then it updates its solution $\sigma(i, j, k)$ by combining it with $\sigma(i, j + 1, k)$.

Lemma 3.7. Let q_k be a rank query. The active copies of q_k in a generic slice HS_i carry enough information to compute the number of items in HS_i strictly smaller than q_k .

Proof. First, if all copies of q_k in slice HS_i are active then the sum of their local solutions $\sigma(i, j, k)$ is exactly the number of items in HS_i strictly smaller than q_k .

Assume, therefore, that not all copies of q_k in slice HS_i are active. Consider the active copy of q_k in the *leading processor* of HS_i with respect to q_k .

- If this copy is active by rule (a4) then its solution $\sigma(i, j, k)$ must be 0, which is the correct number of items in HS_i strictly smaller than q_k .
- If this copy is active by rule (a3) then its solution $\sigma(i, j, k)$ is updated to read $\frac{n}{\sqrt{p}}$, which is the correct number of items in HS_i strictly smaller than q_k .
- If this copy is active by rule (a1) or (a2) then its solution $\sigma(i, j, k)$ is updated to read $\sigma(i, j, k) + (j - 1)\frac{n}{p}$, which is the correct number of items in HS_i strictly smaller than q_k in all processors $P_{i,1}, P_{i,2}, \dots, P_{i,j}$.

It is important to note that the solutions of the other active copies of q_k are *not changed* by the updates. Thus, after the required updates, the collection of active copies of q_k in slice HS_i carry enough information to correctly compute the number of items in HS_i smaller than q_k . The conclusion follows. \square

The next task of Stage 3 is to gather all the active copies of queries to the first row and column of processors $P_{i,0}, P_{0,i}, (1 \leq i \leq \sqrt{p} - 1)$, as illustrated in Figure 3.8. This task can be performed in two gather rounds as follows. In the first round, the gather operations proceed rowwise in parallel in each slice HS_i .

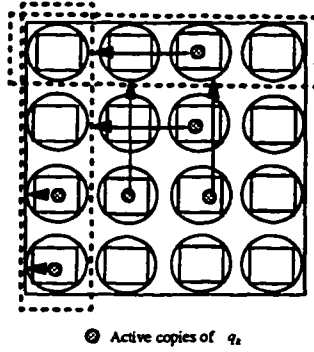


Figure 3.8: *Target of the gather operations in Stage 3*

By Lemma 3.5, no gather operation contains more than m active copies. Since every query is involved in at most one horizontal gather operation, this first round takes $O(T_G(m, \sqrt{p}))$ time. Similarly, the vertical gather operations are performed in parallel taking $O(T_G(m, \sqrt{p}))$ time. In summary, the following result is stated.

Lemma 3.8. The solutions of all active copies of queries in Q can be gathered to the first row and column of processor m per processor in $O(T_G(m, \sqrt{p}))$ time. \square

To complete the algorithm, the various copies of queries in Q moved to the processors will be collected and combined. This can be accomplished in many ways. Only the processing for the first row of processors is explained below; the processing for the first column being is dual.

- A simple All-to-All gather will re-arrange the data in such a way that all copies of a query, q , will be placed in the same processor. Now every processor will, in parallel, compute the final solution. The total time for the process will be $\sqrt{p} + T_{AAG}(m\sqrt{p}, \sqrt{p})$. All-to-All gather operation is generally quite expensive on many platforms. So this approach may not be preferable.
- If $m \leq \frac{M}{\sqrt{p}}$, then by a single gather the whole data can be placed in a single processor. Then either the whole computation can be done in a single processor,

or the copies belonging to different queries can be scattered to different processors, to complete the final computation. This approach will take $O(T_G((m\sqrt{p}, \sqrt{p}) + \min(m\sqrt{p}, T_S(m\sqrt{p}, p))))$.

- If $m > \frac{M}{\sqrt{p}}$, then repeat the gather and scatter process $\frac{\sqrt{p} * m}{M}$, times. The computation time will scale accordingly.

Consequently, the following result is obtained.

Theorem 3.9. An arbitrary instance of the BSR problem involving a sorted matrix of size $\sqrt{n} \times \sqrt{n}$ and a set of m queries, can be solved in $O(T_B(m, p) + (m + \frac{n}{p}) \log(m + \frac{n}{p}) + T_G(m, \sqrt{p}) + \min(T_{AAG}(m\sqrt{p}, \sqrt{p}), \frac{M}{\sqrt{p}} * (T_S(m\sqrt{p}, p))))$ time on the ACM(n,p,M). \square

CHAPTER 4

THE COMPUTATIONAL PARADIGM ON THE MMB

As noted in the discussion of the introduction, this chapter considers applying the computational paradigm in a fine grain scenario; in particular, the focus will be on the Mesh with Multiple Broadcasting. Specifically, the purpose of this chapter is to discuss in detail time optimal solutions for the MQ problem and its instances on the MMB. It will be shown that the knowledge of the communication system will lead to time optimal solutions for some problems.

The remainder of this chapter is organized as follows: Section 4.1 presents the lower bounds; Section 4.2 describes a generic algorithm for the MQ problem. The remaining sections discuss various instances of the MQ problem. Specifically, Section 4.3 discusses rank-related problems; Section 4.4 discusses the multiple point location problem and several of its variants and applications; Section 4.5 addresses proximity-related problems; finally, Section 4.6 discusses the multiple stabbing problem.

4.1 Lower Bounds

The purpose of this section is to establish a non-trivial lower bound for some instances of the MQ problem on meshes with multiple broadcasting. This is achieved by first, proving a lower bound for a different problem, namely the *gather* problem. Once established, this lower bound will be used to derive lower bounds for all the problems of interest.

4.1.1 The Gather Problem

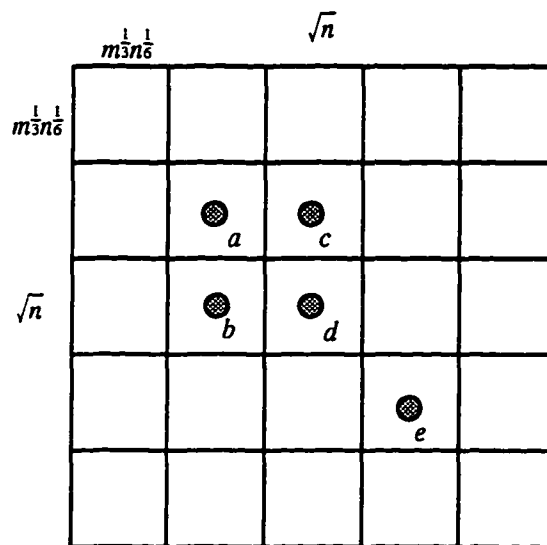


Figure 4.1: *Adversary instance of the gather problem*

An instance of the *gather* problem consists of a set of n items A and of a partition $R = \{A_1, A_2, \dots, A_m\}$ of A . A is pretiled in a MMB one item per processor in an arbitrary fashion. The problem is to gather information about each of the A_i , $1 \leq i \leq m$, in a distinct “target” processor P_i , that is, each processor P_i should know about all the items in A_i . Recall that a processor can only hold a constant amount of information

therefore the gathering operation can only be of an “accumulative” nature. For this operation to be complete P_i should eventually have information about all the items in A_i . For example, the gathering operation may be to find the sum of all elements in A_i .

Lemma 4.1. The lower bound for the gather problem is $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$, given $|A_i| = (\frac{n}{m})^{\frac{2}{3}}$, $1 \leq i \leq m$.

Proof: Consider the mesh as consisting of submeshes of size $s \times s$, where $s = m^{\frac{1}{3}}n^{\frac{1}{6}}$. The proof is based on an adversary argument. The main aim of the adversary is to slow down the progress of the algorithm as much as possible. To this effect, the adversary places one element of each A_i in a submesh: this is possible as $m \leq s^2$ and there are exactly $(\frac{n}{m})^{\frac{2}{3}}$ submeshes. Consider elements $a, b, c, d, e \in A_k$, for some $k \in \{1, 2, \dots, m\}$, as illustrated in Figure 4.1. Using local connections only, the time taken for any processor to know the combined information of a, b, c, d , and e is at least s . It follows that in order to collect the information the bus system must be used. The amount of information that has to be gathered per query is $(\frac{n}{m})^{\frac{2}{3}}$. The total amount of information that needs to be broadcast is $O((\frac{n}{m})^{\frac{2}{3}} * m)$ and there are \sqrt{n} buses. Consequently, the time taken will be in $\Omega(\frac{(\frac{n}{m})^{\frac{2}{3}} * m}{\sqrt{n}})$ which is $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$. \square

4.1.2 Lower bounds for instances of the MQ problem

This subsection describes the lower bounds of some of the instances of the MQ problem. The approach used here is either to prove that the problem is equivalent to the gather problem, or to reduce a problem whose lower bound is known (say PK) to the problem at hand (say PU). To achieve the reduction from PK to PU , the general input to PK is mapped on to the input to PU and the solution of PU will be mapped back to get the solution for PK . If the time for mapping is less

than the lower bound of PK then PU should have the same lower bound as PK .

The following setup forms the general setting for all the subsequent problems. Given a set of items A and a set of queries Q , where $|A| = n$, and $|Q| = m$. The items are pretilled in a MMB of size $\sqrt{n} \times \sqrt{n}$, one per processor. Similarly, the elements of the set Q are placed in the first $\frac{m}{\sqrt{n}}$ columns, one per processor.

The Multiple Rank Problem

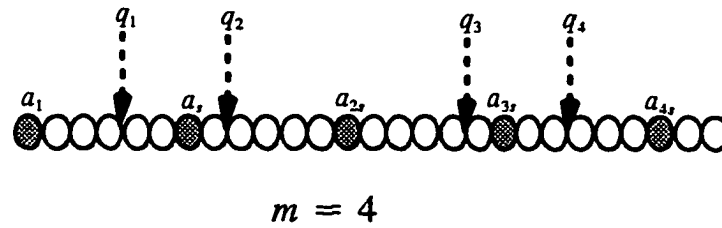


Figure 4.2: Construction for multiple rank problem

It is to be proved that there exists an instance of the multiple rank problem where each query has to gather information about $(\frac{n}{m})^{\frac{2}{3}}$ items. Consider the sorted sequence of elements belonging to A and Q . Let them be $\{a_1, a_2, \dots, a_n\}$ and $\{q_1, q_2, \dots, q_m\}$, respectively. The values of the queries are selected such that, $a_{(i-1)*s+1} \leq q_i \leq a_{i*s}$, where $s = (\frac{n}{m})^{\frac{2}{3}}$. Note that, since $m \leq n$,

$$(\frac{n}{m})^{\frac{2}{3}} * m = n^{\frac{2}{3}} * m^{\frac{1}{3}} \leq n^{\frac{2}{3}} * n^{\frac{1}{3}} = n.$$

Refer to Figure 4.2, for an illustration of the placement of queries for $m = 4$.

It is clear from the construction that each query has to learn information about at least s items independent of the other queries. This instance of the multiple rank problem is equivalent to the gather problem. Thus, the following lemma is obtained.

Lemma 4.2. The multiple rank problem has a lower bound of $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$. \square

The same lower bound holds for a variant of the multiple rank problem. Instead of the rank of a query q , identify the two consecutive items a_i, a_{i+1} which belong to the sorted sequence of A such that $a_i < q \leq a_{i+1}$. This variant of the multiple rank problem is used to prove lower bounds for some problems, and it will be referred to as the *bracketing* variant of the multiple rank problem.

The Histogram Problem

The proof of the lower bound for the histogram problem is similar to the multiple rank problem, with the frequency of each level $q_i \in Q$, $1 \leq i \leq m$, being required to equal $(\frac{n}{m})^{\frac{2}{3}}$. Thus, the following lemma is obtained.

Lemma 4.3. The histogram problem has a lower bound of $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$. \square

The Multiple Point Location Problem

In this problem, it is necessary that if the query point is outside the convex hull of A , the tangents from the point to convex hull are returned.

Let the convex hull of a set, A , of points in the plane be denoted by $CH(A)$. Furthermore, let $CH(A) = \{a_1, a_2, \dots, a_n\}$ where (a_i, a_{i+1}) is an edge of $CH(A)$, $1 \leq i \leq n$ (for convenience assume $a_0 = a_n$). This lower bound is based on a construction. Here the following instance of the multiple point location problem is of interest. Let all the elements of A belong to the convex hull $CH(A)$. Consider a sample formed by taking every s^{th} point of $CH(A)$, refer to Figure 4.3. The idea is to place each query in such a way that the solutions to any two queries will have to be determined independently. This is achieved by placing each q_i in the triangular region determined by the edge $(a_{(i-1)*s}, a_{i*s})$ of the sample polygon and the two lines determined by the points $\{a_{(i-1)*s-1}, a_{(i-1)*s}\}$ and $\{a_{i*s}, a_{i*s+1}\}$, $1 \leq i \leq m$. (Note: Subscripts which are negative or greater than n are treated in a modulo fashion.) It is clear from the construction that each query has to learn information about

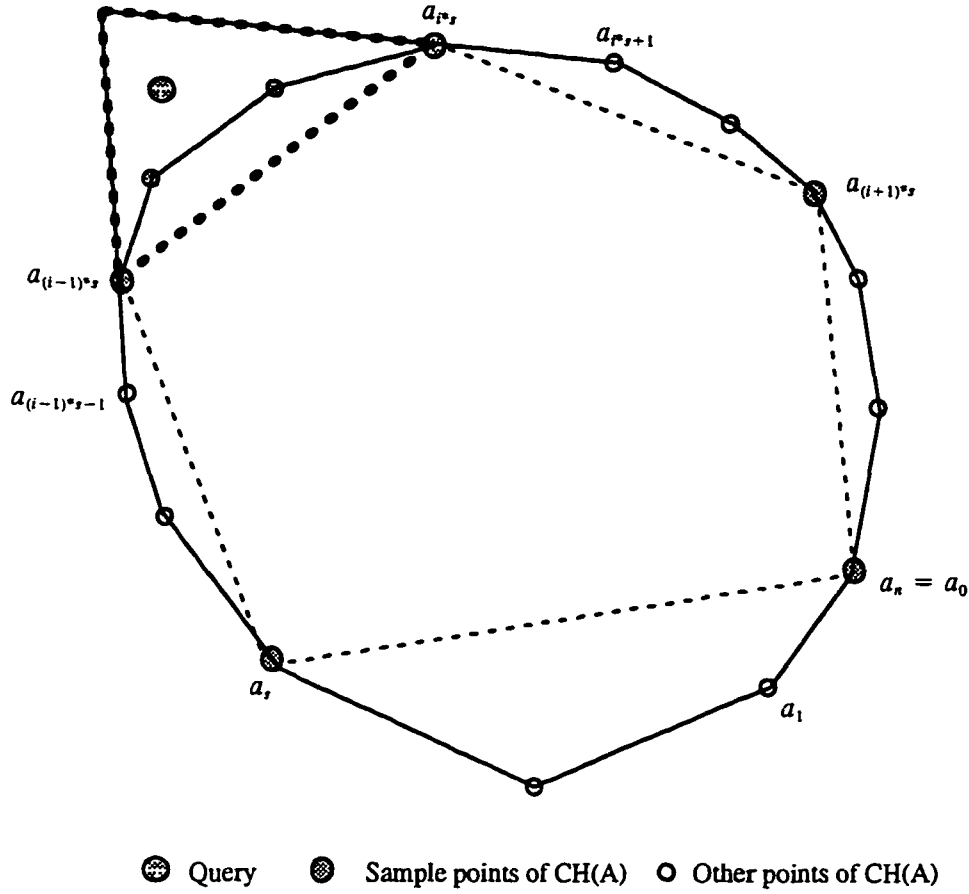


Figure 4.3: *Construction for multiple point location problem*

at least s points of the $CH(A)$ to determine its solution independent of the other queries. Thus, the following lemma is obtained.

Lemma 4.4. The multiple point location problem has a lower bound of $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$.

□

It is noted that the lemma also follows by reducing the bracketing variant of the multiple rank problem to the multiple point location problem. This can be achieved by converting items and queries to polar coordinates as follows. Let d be the element larger than every item and every query, assume that all the items and queries are positive; $a \in A$ is mapped to $(r, \frac{a}{d})$ for some fixed r , and $q \in Q$ is mapped

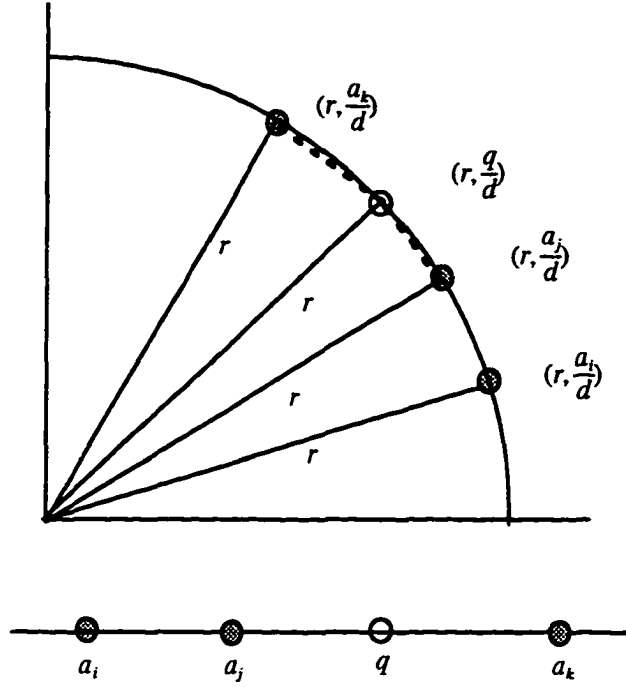


Figure 4.4: *Reduction for multiple point location problem*

to $(r, \frac{q}{d})$. This will guarantee that all the queries are outside the convex hull of the items and their tangents can be used to determine the solution to the variant of the multiple rank problem in the obvious way. In the Figure 4.4, the mapping of items and queries to points is depicted. Specifically, there are three items, a_i, a_j, a_k , and a query, q , such that $a_i < a_j < q < a_k$. The point $(r, \frac{q}{d})$, corresponding to q has $(r, \frac{a_k}{d})$ and $(r, \frac{a_j}{d})$ as its tangency points indicating that the solution for query q consists of the points a_k, a_j .

The Containment Problem

The construction for this problem is same as the one illustrated in Figure 4.3 above. Note that to determine the solution, each query has to check if it is either inside or outside convex hull of A . If the skips one query then by the construction it could be placed either inside or outside the convex hull of A and invalidate the

answer. Thus, the following lemma is obtained.

Lemma 4.5. The containment problem has a lower bound of $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$. \square

The Multiple Closest Segment Problem

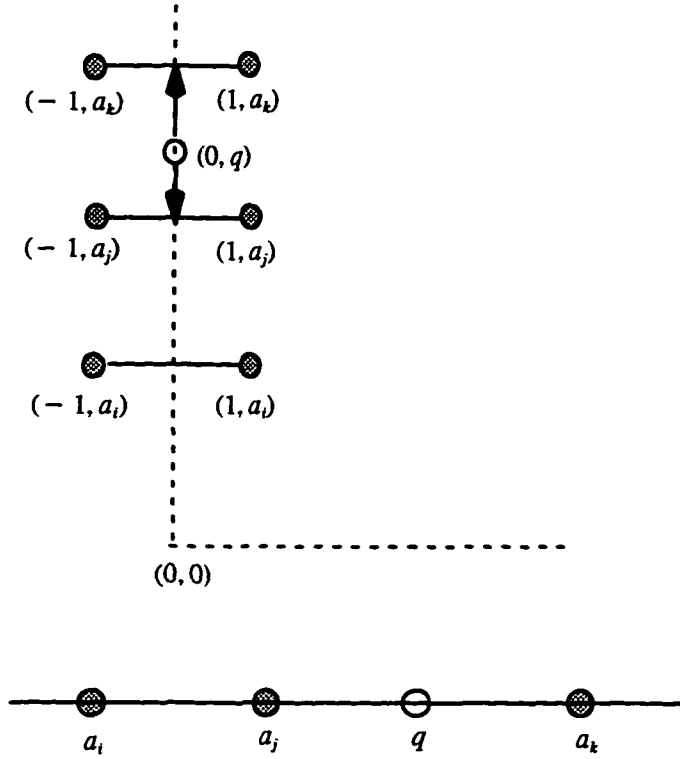


Figure 4.5: *Reduction for multiple closest segment problem*

Again the lower bound is proved by reducing the bracketing variant of the multiple rank problem to the multiple closest segment problem. Consider the input to the bracketing variant of the multiple rank problem, items A and queries Q , generate input to the multiple closest segment problem by mapping each item $a \in A$ to the segment $((-1, a), (1, a))$ and each query $q \in Q$ to the point $(0, q)$. The segments returned by each query will correspond to the solution of the variant of the multiple rank problem. In Figure 4.5, the mapping from items to segments

and queries to points, is illustrated. Here items a_i, a_j, a_k and query, q are mapped to segments $((-1, a_i), (1, a_i))$, $((-1, a_j), (1, a_j))$, $((-1, a_k), (1, a_k))$, and point $(0, q)$, respectively. Note that, $a_i < a_j < q < a_k$, and the solution to point $(0, q)$ is the segment pair $(((-1, a_j), (1, a_j)), ((-1, a_k), (1, a_k)))$ indicating that solution to the query q is the item pair (a_j, a_k) . Thus, the following lemma is obtained.

Lemma 4.6. The multiple closest segment problem has a lower bound of $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$.

□

The Multiple Range Problem

It is easy to see that an instance of the multiple range problem can be generated where each query rectangle contains $(\frac{n}{m})^{\frac{2}{3}}$ items, thus forcing each rectangle to gather information about $(\frac{n}{m})^{\frac{2}{3}}$ items independently, implying that this instance of the multiple range problem is equivalent to the gather problem. Thus, the following lemma is obtained.

Lemma 4.7. The multiple range problem has a lower bound of $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$. □

The Multiple Circle Problem

This lower bound proof proceeds in the same lines as the lower bound proof of multiple range problem. It is easy to see that an instance of the multiple circle problem can be generated where each query circle contains $(\frac{n}{m})^{\frac{2}{3}}$ items. Thus, forcing each circle to gather information about $(\frac{n}{m})^{\frac{2}{3}}$ items independently, implying that this instance of the multiple circle problem is equivalent to the gather problem. Thus, the following lemma is obtained.

Lemma 4.8. The multiple circle problem has a lower bound of $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$. □

The Multiple Stabbing Problem

The multiple rank problem can be reduced to the multiple stabbing problem. Given items A and queries Q for the multiple rank problem, generate an instance

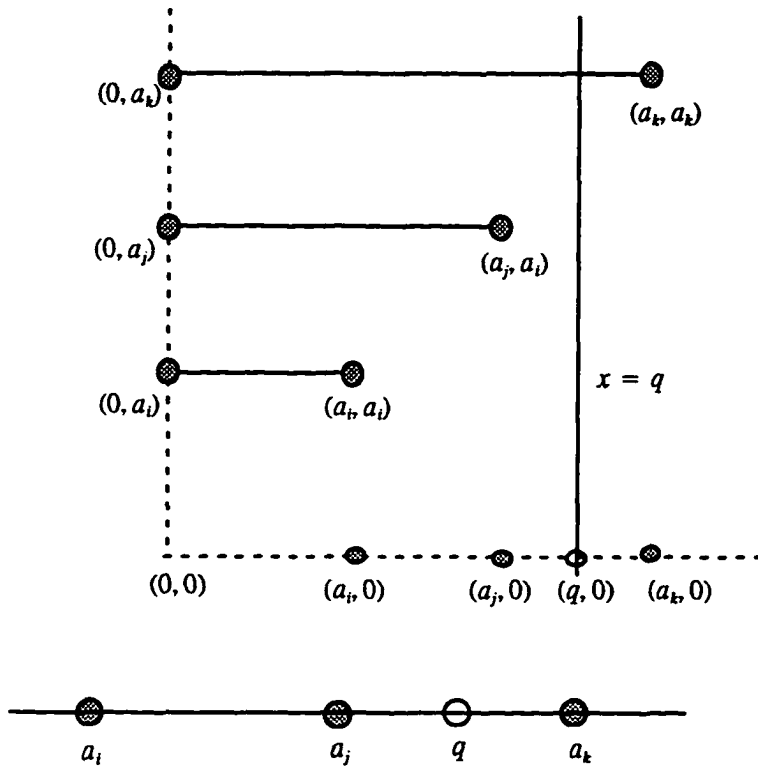


Figure 4.6: *Mapping for multiple stabbing problem*

of the multiple stabbing problem by mapping each item $a \in A$ to the segment $((0, a), (a, a))$, and each query $q \in Q$ to the line $(x = q)$ (assume that all the items and queries are distinct). Let the number of line segments intersected by the line corresponding to q be k , it's clear that the rank of q is $n - k$. This provides a solution to the multiple rank problem. In Figure 4.6, the mapping from items to segments and queries to lines, is illustrated. Similar to the multiple closest segment problem, items a_i, a_j, a_k and query q are mapped to segments $((0, a_i), (a_i, a_i))$, $((0, a_j), (a_j, a_j))$, $((0, a_k), (a_k, a_k))$, and line $x = q$, respectively. Again, $a_i < a_j < q < a_k$, and the line $x = q$ intersects only those segments $((0, a), (a, a))$ where $a > q$, here in particular, $x = q$ intersects only $((0, a_k), (a_k, a_k))$. This indicates that solution to the query q is obtained directly from solution to the line $x = q$. Thus, the following

lemma is obtained.

Lemma 4.9. The multiple stabbing problem has a lower bound of $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$. \square

The Multiple Closest Point Problem

When the data for this problem is restricted to one dimension, it is precisely the variant of the multiple rank problem. Therefore the lower bound for the multiple rank problem should hold for this problem. Thus, the following lemma is obtained.

Lemma 4.10. The multiple closest point problem has a lower bound of $\Omega(m^{\frac{1}{3}}n^{\frac{1}{6}})$.

\square

4.2 A Generic Multiple Query Algorithm on MMB

Most of the algorithms are similar to the generic ones developed for the ACM, so only the MMB specific portions of the algorithm are presented in this section. Recall, a generic instance of the MQ problem involves four parameters A , Q , ϕ , and f . Recall further that for every query q_i ($1 \leq i \leq m$), let $S_i = \{a_j \in A \mid \phi(q_i, a_j) = \text{"yes"}\}$ and that the *solution* of q_i is $f(S_i)$. Next assume that the set A is stored in some order, one item per processor, in \mathcal{R} a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Further assume the set Q is stored in the first $\frac{m}{\sqrt{n}}$ columns of \mathcal{R} , one query per processor. To make the notation less cumbersome, let[†]

$$s = m^{\frac{1}{3}}n^{\frac{1}{6}}. \quad (4.1)$$

Note that \mathcal{R} can be viewed as consisting of submeshes $R_{i,j}$ ($1 \leq i, j \leq \frac{\sqrt{n}}{s}$), of size $s \times s$, with $R_{i,j}$ involving processors $P(r, c)$ with $1 + (i-1)s \leq r \leq is, 1 + (j-1)s \leq c \leq js$. Occasionally, it will be convenient to view the mesh \mathcal{R} as consisting of submeshes $S_1, S_2, \dots, S_{\frac{\sqrt{n}}{s}}$ of size $s \times \sqrt{n}$, with S_i ($1 \leq i \leq \frac{\sqrt{n}}{s}$) comprising of the

[†]For simplicity assume that s , $\frac{\sqrt{n}}{s}$, and $\frac{m}{s}$ are integers.

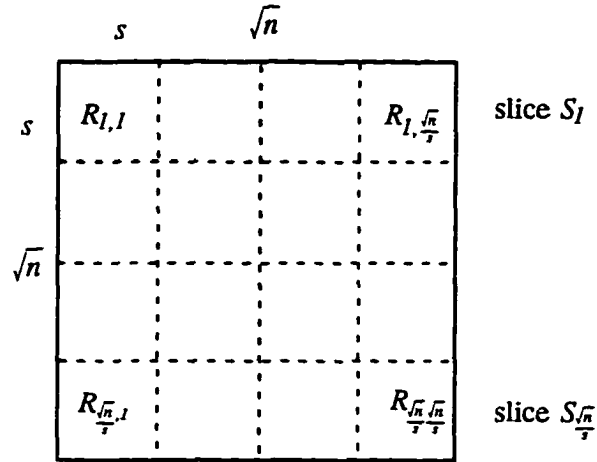


Figure 4.7: *Partition into submeshes $R_{i,j}$*

submeshes $R_{i,1}, R_{i,2}, \dots, R_{i,\frac{\sqrt{n}}{s}}$. Every such submesh S_i will be referred to as a *slice* of \mathcal{R} . For an illustration, refer to Figure 4.7.

Just as in the generic algorithm for the ACM, there are three stages in this algorithm. The remainder of this section is devoted to a detailed description of the computation that takes place in each of these stages.

Stage 1.

The purpose of this stage is to replicate the set Q of queries into the leftmost $\frac{m}{s}$ columns of each submesh $R_{i,j}$. The plan is to move the queries in every column k , ($1 \leq k \leq \frac{m}{\sqrt{n}}$), of \mathcal{R} into columns $(k-1)\frac{\sqrt{n}}{s} + 1$ through $k\frac{\sqrt{n}}{s}$ of each submesh $R_{i,j}$. To begin, every processor $P(r, k)$ ($1 \leq r \leq \sqrt{n}$) broadcasts the query it holds horizontally to processor $P(r, r)$. In turn, processor $P(r, r)$ broadcasts the query received vertically to processors $P(ts + 1 + (r-1) \bmod s, r)$ ($0 \leq t \leq \frac{\sqrt{n}}{s} - 1$).

As noted before, as a result of this data movement, the queries originally stored in column k of \mathcal{R} have been replicated in the diagonal processors of the submeshes in every slice. From now on, every slice is processed in parallel. Specifically,

the queries stored by the diagonal processors of $R_{i,1}$ are replicated, using the row buses in slice S_i , into the $(k-1)\frac{\sqrt{n}}{s} + 1$ -th column of each $R_{i,j}$ in slice S_i . Next, the queries stored by the diagonal processors in $R_{i,2}$ are replicated, using the row buses in slice S_i , into the $(k-1)\frac{\sqrt{n}}{s} + 2$ -th column of each $R_{i,j}$, and so on.

It is easy to see that the task of replicating the queries in one column of \mathcal{R} takes $O(\frac{\sqrt{n}}{s})$ time. Therefore, as long as $m \geq \sqrt{n}$, the queries initially stored in the leftmost $\frac{m}{\sqrt{n}}$ columns of \mathcal{R} can be replicated in time $O(\frac{\sqrt{n}}{s} * \frac{m}{\sqrt{n}}) = O(\frac{m}{s}) = O(\frac{m^{\frac{2}{3}}}{n^{\frac{1}{6}}}) \subseteq O(m^{\frac{1}{3}}n^{\frac{1}{6}})$. In case $m < \sqrt{n}$, the queries are replicated in a way similar to the one described. The complexity of this data movement is, again, $O(\frac{m}{s}) \subseteq O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time.

With this, the goal of Stage 1 has been achieved: the queries have been replicated into each of the submeshes $R_{i,j}$. Thus, the following result is obtained.

Lemma 4.11. The set Q of queries initially stored in the first $\frac{m}{\sqrt{n}}$ columns of \mathcal{R} can be replicated into the first $\frac{m}{s}$ columns of each $R_{i,j}$ in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. \square

Stage 2.

In Stage 2, to avoid broadcasting conflicts the bus system is ignored, and each submesh $R_{i,j}$ will act as an unenhanced mesh. The way the local instance of the MQ problem is solved in each $R_{i,j}$ is application-dependent. It is assumed that this stage can be performed in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time.

Stage 3.

At the end of Stage 2, every processor of each submesh $R_{i,j}$ that stores a query q_u will store its local solution $f(S_u)$. The goal of Stage 3 is to combine these local solutions into the solution of q_u in the original instance of the MQ problem. Once the processing in Stage 3 is complete, the solution of the original instance of the MQ problem has been obtained.

In preparation for this, the first goal of this stage is to arrange the ordered pairs $(q_u, f(S_u))$ in column-major order in the leftmost $\frac{m}{s}$ columns of every submesh $R_{i,j}$, sorted by u , the index of their first component. Recall that by using an optimal sorting algorithm [47, 83], this goal can be achieved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time.

From now on, the processing depends on whether or not $m \geq n^{\frac{1}{4}}$.

Case 1. $m \geq n^{\frac{1}{4}}$.

In this case, in every $R_{i,j}$ there is at least one full column of queries. The various slices of \mathcal{R} are processed in parallel. For illustration purposes, processing that takes place in slice S_i is detailed. Let $(q_u, f(S_u))$ be a generic query-solution pair stored by a processor $P(r, c)$ in $R(i, 1)$. By virtue of the data movement described in the preamble to this stage, a similar pair is stored by processors $P(r, c + ts)$ in $R_{i,t+1}$, for $1 \leq t \leq \frac{\sqrt{n}}{s} - 1$. In $\frac{\sqrt{n}}{s} - 1$ time units, sequentially, every processor $P(r, c + ts)$ broadcasts to $P(r, c)$ the second component of the pair $(q_u, f(S_u))$ it holds. It is easy to see that in $O(\frac{\sqrt{n}}{s})$ time, $P(r, c)$ can accumulate the solutions of q_u in the whole slice S_i . Since, S_i has s buses, entire columns of queries can be processed in this way. Consequently, the process of accumulating the corresponding solutions for all the queries can be done, in each slice, in $O(\frac{\sqrt{n}}{s} * \frac{m}{s}) = O(\frac{m\sqrt{n}}{s^2}) = O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time.

Finally, after transposing the first $\frac{m}{s}$ columns into rows in each $R_{i,1}$ the above process can be repeated in the vertical slice consisting of the submeshes $R_{1,1}, R_{2,1}, \dots, R_{\frac{\sqrt{n}}{s},1}$ thus accumulating for every query the corresponding solutions in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time.

Case 2. $m < n^{\frac{1}{4}}$.

In this case, the queries in each $R_{i,j}$ occupy only a segment of the first column, as illustrated in Figure 4.8(a). To simplify, it is assumed without loss of generality that for some positive integer c , $s = c * m$. Using local connections only, the m

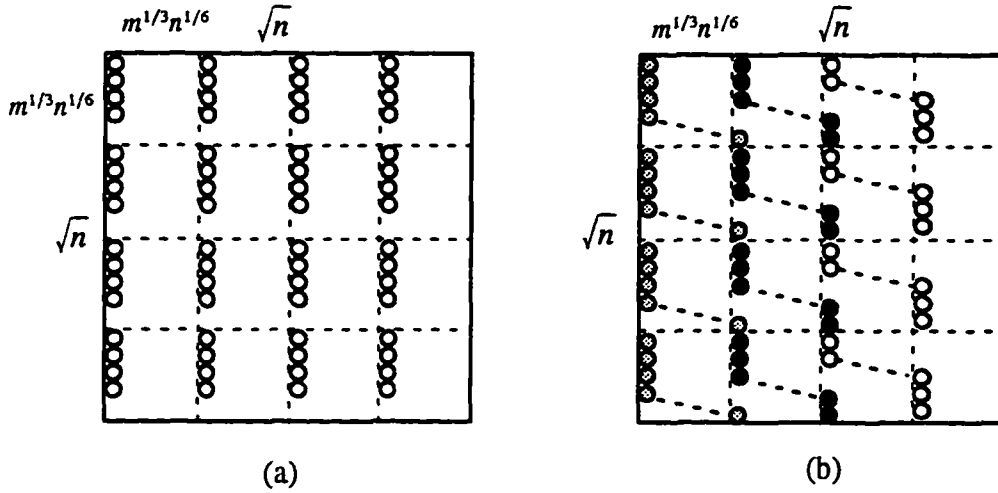


Figure 4.8: *Data movement of Case 2*

queries in $R_{i,j}$ ($1 \leq j \leq \frac{\sqrt{n}}{s}$) will be moved vertically, in lock step, into positions $[(j-1) \bmod c] * m + 1$ through $[(j-1) \bmod c] * m + m$ in the first column of $R_{i,j}$. Clearly, this operation takes no more than $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. For an illustration, refer to Figure 4.8.

The consecutive groups of c of the $R_{i,j}$'s in slice S_i is referred to as a *run*. In Figure 4.8(b), various runs are depicted using different shades of gray. The motivation for this terminology comes from the observation that by virtue of the previous data movement, the queries in each run occupy distinct rows. Using horizontal buses in S_i , the queries can be moved in parallel in $O(1)$ time. Specifically, the intention is to move the queries in S_i into the columns of $R_{i,1}$. It is easy to confirm that there are exactly $\frac{\sqrt{n}}{s} * \frac{m}{s} = \frac{m\sqrt{n}}{s^2} = m^{\frac{1}{3}}n^{\frac{1}{6}}$ runs, and so the operation of compacting these runs into $R_{i,1}$ will take $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time.

Next, sort the queries in each submesh $R_{i,1}$ ($1 \leq i \leq \frac{\sqrt{n}}{s}$) in row-major order by query index. This data movement guarantees that the solutions corresponding to the same query will occur next to one another. Proceeding row by row, these

solutions are accumulated and stored in the leftmost processor in each row. Note that no such processor can contain accumulated results pertaining to more than two distinct queries. Proceeding vertically, the final sums are accumulated for every distinct query in every submesh $R_{i,1}$. Now transposing columns into rows and shifting appropriately, horizontal runs are created which will be compacted in $R_{1,1}$. Here the queries are sorted again and, as before, the partial results are accumulated.

Thus, the entire computation in Stages 1–3 can be performed in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time and the following result is obtained.

Theorem 4.12. Provided that every local instance of the MQ problem in Stage 2 can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time, the original instance of the MQ problem involving a set of n items and a set of m queries can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. \square

In each of the remaining sections, the details of Stage 2 of the algorithms are discussed.

4.3 Rank-Related Computations

4.3.1 The Multiple Rank Problem

As in Chapter 2, multiple rank problem will be referred to as MULTI-RANK. For definiteness, both the items $A = \{a_1, a_2, \dots, a_n\}$ in the database and the queries in $Q = \{q_1, q_2, \dots, q_m\}$ are assumed to come from a totally ordered universe.

Stage 2. To avoid broadcasting conflicts, in Stage 2 the bus system is ignored, and every submesh $R_{i,j}$ will act as an unenhanced mesh. The processing of Stage 2 will be partitioned into two substages, each solving a different instance of the MQ problem. To define precisely the local instances of the MQ problem that are solved,

begin by sorting the items in every $R_{i,j}$ in row-major order. To simplify the notation, let b_1, b_2, \dots, b_{s^2} stand for the sorted sequence of items in $R_{i,j}$. In this notation, the last column of $R_{i,j}$ contains, in top-down order, the items $b_s, b_{2s}, \dots, b_{s^2}$. Now the instance of the MQ problem that is solved in $R_{i,j}$ has the following parameters:

- the (sorted) sequence $B_{i,j} = b_s, b_{2s}, \dots, b_{s^2}$,
- the set Q of queries,
- a decision problem $\phi : Q \times B_{i,j} \rightarrow \{\text{"yes"}, \text{"no"}\}$ such that for every $1 \leq u \leq m$ and $b \in B_{i,j}$, $\phi(q_u, b) = \text{"yes"}$ if and only if $b < q_u$, and
- for every query q_u , let S_u be the set of items b in $B_{i,j}$ for which $\phi(q_u, b) = \text{"yes"}$. Let $f(S_u) = |S_u| + 1$ or, equivalently, the unique value k for which $b_{(k-1)s} < q_u \leq b_{ks}$ if $q_u \leq b_{s^2}$, and $s + 1$ otherwise. (To handle boundary conditions, let $b_0 = -\infty$.)

Substage 2.1. The purpose of Substage 2.1 is to solve in every $R_{i,j}$ the instance of the MQ problem that was defined above. For every query q_u , $f(S_u)$ will be referred to as the *row rank* of q_u in $R_{i,j}$. To accomplish the task specific to this stage, the last column of $R_{i,j}$ is replicated in all the columns of the submesh. This is done in the obvious way in $O(s) = O(m^{\frac{1}{3}} n^{\frac{1}{6}})$ time.

Next, in each of the leftmost $\frac{m}{s}$ columns of $R_{i,j}$, the items received are permuted vertically, in lock step, in such a way that in $O(m^{\frac{1}{3}} n^{\frac{1}{6}})$ time every query meets every one of the items $b_s, b_{2s}, \dots, b_{s^2}$. As a consequence, every query has enough information to compute its row rank. Thus, the following result can be stated.

Lemma 4.13. The row ranks of all queries in every submesh $R_{i,j}$ can be determined in $O(m^{\frac{1}{3}} n^{\frac{1}{6}})$ time. \square

Substage 2.2. The goal of this stage is to solve another instance of the MQ problem in each submesh $R_{i,j}$. More specifically, this instance involves the following parameters:

- the sorted sequence $A_{i,j} = b_1, b_2, \dots, b_{s^2}$,
- the set Q of queries,
- a decision problem $\phi : Q \times A_{i,j} \rightarrow \{\text{"yes"}, \text{"no"}\}$ such that for every $1 \leq u \leq m$ and $b \in A_{i,j}$, $\phi(q_u, b) = \text{"yes"}$ if and only if $b < q_u$, and
- for every u ($1 \leq u \leq m$), let S_u be the set of items b in $A_{i,j}$ for which $\phi(q_u, b) = \text{"yes"}$. Let $f(S_u) = |S_u|$.

The solution of this instance of the MQ problem will use as a stepping stone the solution of the instance of the MQ problem solved in Substage 2.1.

By using an optimal sorting algorithm for meshes [47, 59, 83], the sequence of queries in each submesh $R_{i,j}$ is sorted in row-major order by row rank in the first $\frac{m}{s}$ rows of the submesh. Each of the first $\frac{m}{s}$ rows of $R_{i,j}$ will be termed a *query-row*. (Recall that every processor in the first $\frac{m}{s}$ rows of $R_{i,j}$ contain one item and one query.) A query-row of $R_{i,j}$ is called *pure* if all the queries in the row share the same row rank. Otherwise, the query-row is termed *impure*. The identification of every query-row as pure or impure follows. Every processor in the last column of $R_{i,j}$ sends the row rank of the query it holds horizontally using local movement only. Upon receiving this information, every processor in the first column has enough information to determine whether its query-row is pure or impure.

Note that for all queries whose row rank is $s+1$, the solution is s^2 . Next, consider the processing for pure query-rows with row ranks at most s . Let r_1, r_2, \dots, r_t

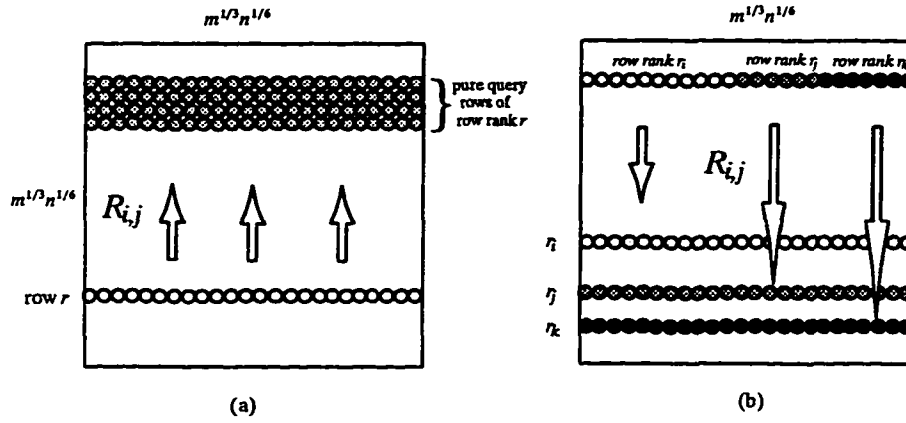


Figure 4.9: *Processing of pure and impure query-rows*

be the row ranks of the pure query-rows in $R_{i,j}$. By moving all the rows of $R_{i,j}$ vertically, in lock step, row r_k will be replicated in every pure query-row with row rank r_k . For an illustration, refer to Figure 4.9(a). Clearly, this data movement takes $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. Further, in every pure query-row, the items are moved horizontally, in lock step. This data movement ensures that every query in a pure query-row determines its solution in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time.

Impure query-rows are handled differently, refer to Figure 4.9(b). Let r be an arbitrary impure query-row of $R_{i,j}$ and let r_1, r_2, \dots, r_t be the row ranks of the queries in row r . Since the sequence of queries was sorted in row-major order, the queries having the same row rank occur consecutively in r . It is important to note that for any p ($1 \leq p \leq s$), at most two impure query-rows contain queries whose row rank is p . In a first step, all queries in impure query-rows are moved to the row of the mesh that equals their row rank. This is done by moving, in lock step, all the query-rows of the mesh vertically. It is easy to confirm that after $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time units, all queries in impure query-rows have reached the row of the mesh that equals their row rank. The previous observation guarantees that no processor stores, as

a result of this data movement, more than *two* queries. In each row, the solution for each query is determined by sending all the items in that row, in lock step, horizontally throughout the row. This ensures that every query in the row finds its solution in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. In summary, the following result can be stated.

Lemma 4.14. The task of solving the queries in every $R_{i,j}$ can be performed in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. \square

The prior results can now be combined to yield the stated goal.

Theorem 4.15. An arbitrary instance of the MULTI-RANK problem involving a set of n items and a set of m queries can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal. \square

4.3.2 Histogram Computation

The algorithm for HISTOGRAM is identical to the algorithm for MULTI-RANK except for a post-processing step that is now described. Let $\text{rank}(q_1), \text{rank}(q_2), \dots, \text{rank}(q_m)$ be the ranks of the queries returned by MULTI-RANK applied to the instance of the HISTOGRAM problem. Now for every i ($1 \leq i \leq m-1$) the solution to q_i is $\text{rank}(q_{i+1}) - \text{rank}(q_i)$, in other words, the number of pixels in A having a grey-level intensity equal to q_i . Furthermore, the solution to q_m is $n - \text{rank}(q_{m-1})$. In summary, the findings are stated by the following result.

Theorem 4.16. An arbitrary instance of the HISTOGRAM problem involving an m -level image of size $\sqrt{n} \times \sqrt{n}$ can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal. \square

4.4 The Multiple Point Location Problem

This section provides with an algorithm for MULTI-LOCATION, and presents a few details of related problems: CONTAINMENT, SEPARABILITY, COMMON-TANGENTS.

It is noted that if the convex hull of A is known, then the MULTI-LOCATION problem can be solved in $O(\sqrt{m})$ time by using the algorithm of Bhagavathi *et al.* [13]. However, just computing the convex hull of n points is known to take $\Omega(\sqrt{n})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. One of the contributions of this work is to show that the MULTI-LOCATION problem as well as the three other problems mentioned can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time without computing the convex hull of A . Refer to Chapter 2, for the formulation of MULTI-LOCATION as an instance of the MQ problem and some of the important geometric preliminaries. As a technicality, for all i ($1 \leq i \leq m$), set S_i to the empty set. Recall that, S_i is the set of all “yes” instances for a query q_i . Also, Stage 3 of the algorithm relies on the same results obtained for the problem in Chapter 2. Stage 2 of the MULTI-LOCATION algorithm is as follows.

Stage 2. Begin by computing the convex hull of the subset $A_{i,j}$ of A in each $R_{i,j}$. This task can be performed in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time using an optimal convex hull algorithm for unenhanced meshes [42]. For simplicity of exposition, it is assumed that the convex hull of $A_{i,j}$ is the convex polygon $P = p_1, p_2, \dots, p_s$ stored in row-major order in $R_{i,j}$. The task specific to Stage 2 is to determine for every point in Q whether it is interior to any of the convex hulls local to the $R_{i,j}$ ’s. Clearly, a query point that is interior to any such convex hull lies in the interior of the convex hull of A and its corresponding solution is the empty set.

The task of determining the queries in Q that are interior to P (i.e., the

convex hull of $A_{i,j}$) will be partitioned into two substages.

Substage 2.1. This stage solves the (simpler) problem of finding lines of support for those points in Q that are exterior to the convex polygon $P' = p_s, p_{2s}, \dots, p_{s^2}$ consisting of the vertices of P whose subscripts are multiples of s , as illustrated in Figure 4.10(a).

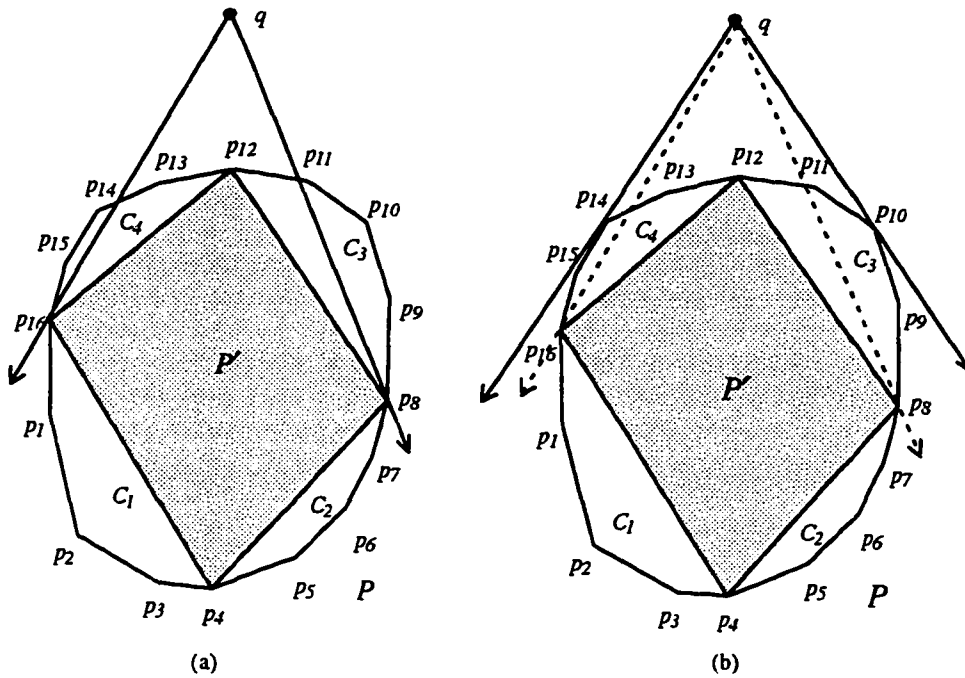


Figure 4.10: Stage 2 of MULTI-LOCATION

Note that P' partitions the boundary of P into chains C_1, C_2, \dots, C_s such that $C_k = p_{(k-1)s+1}, p_{(k-1)s+2}, \dots, p_{ks}$ ($1 \leq k \leq s$).

The vertices of P' are stored by the processors in the last column of $R_{i,j}$, and that every chain C_k ($1 \leq k \leq s$) defined above involves points stored by processors in row k of the submesh. As in Stage 2 of the MULTI-RANK algorithm, the contents of the last column of each $R_{i,j}$ is replicated in all the columns of the submesh. In each of the first $\frac{m}{s}$ columns, the queries are moved vertically, in lock step, in such

a way that in $2m^{\frac{1}{3}}n^{\frac{1}{6}}$ time units every query meets every point of P' and thus can determine the two supporting lines to P' , if they exist. To clarify this last point, note that, for every pair (q_u, p_{ks}) , whether the line determined by q_u and p_{ks} is a supporting line for P' can be determined in $O(1)$ time by checking if both $p_{(k-1)s}$ and $p_{(k+1)s}$ are to the same side of the line.

Substage 2.2. The purpose of this substage is to use the partial solution obtained in Substage 2.1 to determine for every point in Q whether it is interior or exterior to P . Additionally, for those points which are exterior to P , the two supporting lines are produced.

Observe that if a supporting ray for P' determined by some point q in Q and some point p_{ks} is a supporting ray for P , then no further action is needed. Otherwise, it is easy to see that the ray qp_{ks} intersects *precisely* one of the chains C_{k-1} or C_k . (To handle boundary conditions let $C_0 = C_s$.) Furthermore, the chain intersected by the ray qp_{ks} can be determined in $O(1)$ time by checking the edges of P incident to p_{ks} . Refer to Figure 4.10(b) for an illustration.

In what follows, the task of determining the left supporting ray is described. Determining the right supporting ray is similar. Consider the points q in Q for which the left supporting ray for P' is not a supporting ray for P . For every such point q , its *chain rank* is defined to be the subscript of the chain intersected by the left supporting ray from q to P' .

Further, the query points in Q are sorted in row-major order by their chain ranks. Assume without loss of generality that the chain rank of q is k . In order to find a left supporting ray for P emanating from q only the left supporting ray for the convex polygon obtained by adjoining to $C_k = p_{(k-1)s+1}, p_{(k-1)s+2}, \dots, p_{ks}$ vertex $p_{(k-1)s}$ is necessary (to handle boundary conditions assume $p_0 = p_{s^2}$). This

can be done in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time by a slight modification of the processing of Stage 2 of the MULTI-RANK algorithm of Section 4.3. Consequently, the following result is proved.

Lemma 4.17. For every query point in Q , the supporting lines to the convex hull of the subset of A in each $R_{i,j}$ can be determined in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. \square

To summarize the findings the following result can be stated.

Theorem 4.18. An arbitrary instance of the MULTI-LOCATION problem involving sets A and Q of cardinalities n and m , respectively, can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal. \square

The three problems mentioned in the beginning of this section are solved by using essentially the same techniques as discussed in Section 2.3 of Chapter 2. In consequence, the following results are stated.

Theorem 4.19. An arbitrary instance of the CONTAINMENT problem involving sets A and Q of cardinalities n and m , respectively, can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal. \square

Theorem 4.20. An arbitrary instance of the SEPARABILITY and COMMON-TANGENTS problems involving sets A and Q of cardinalities n and m , respectively, can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. \square

4.5 Proximity-Related Computations

The purpose of this section is to show that four fundamental problems in pattern recognition, robotics, and image processing can be solved elegantly by stating them

as special instances of the MQ problem. In each case, only the application specific requirements of Stage 2 are considered as Stages 1 and 3 progress as previously described.

4.5.1 The Multiple Closest Segment Problem

Stage 2. The processing of Stage 2 will use the algorithm of Jeong and Lee [42] that provide an optimal solution to the CLOSEST-SEGMENT on unenhanced meshes. Using this algorithm the local instance in every $R_{i,j}$ is solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time, which leads to the following result.

Consequently, the following result is obtained.

Theorem 4.21. An arbitrary instance of the CLOSEST-SEGMENT problem involving a set of n non-intersecting line segments and a set of m points in the plane, can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal. \square

4.5.2 The Multiple Range Problem

Stage 2. The processing of Stage 2 will use the algorithm for CLOSEST-SEGMENT, as described in Section 2.4 of Chapter 2. Using the algorithm in [42] the local instance of CLOSEST-SEGMENT problem in each $R_{i,j}$ is solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. Consequently, the following result is obtained.

Theorem 4.22. An arbitrary instance of the MULTI-RANGE problem involving a set A of n points in the plane and a set Q of m non-overlapping rectangles, can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal. \square

4.5.3 The Multiple Circles Problem

The MULTI-CIRCLE problem is solved using the same technique as developed in Chapter 2, which can be formalized in this result.

Theorem 4.23. An arbitrary instance of the MULTI-CIRCLE problem involving a set A of n points in the plane and a set Q of m disjoint circles, can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal. \square

4.5.4 The Multiple Closest Point Problem

Stage 2. The processing of Stage 2 will be partitioned into two substages. In the first stage, construct the Voronoi diagram of the points of A located in every submesh $R_{i,j}$. This first task can be performed in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time using the algorithm of Jeong and Lee [42] for unenhanced meshes.

In the second substage, identify for every point in Q the Voronoi polygon that contains it. Once the identity of the enclosing Voronoi polygon is known, the local instance of the CLOSEST-POINT problem is, essentially, solved. The problem at hand can be solved efficiently by observing that the total number of edges of the Voronoi diagram of the subset of point of A located in $R_{i,j}$ is in $O(s^2)$ and that, consequently, these edges can be stored at most one per processor in every $R_{i,j}$.

A further key observation is that to identify, for every point q in Q the unique enclosing Voronoi polygon, it is sufficient to identify the first Voronoi edge intersected by a ray originating at q and going in the positive y -direction. This is, of course, an instance of the CLOSEST-SEGMENT problem which can be solved in each $R_{i,j}$ in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. Combining these parts yields the following result.

Theorem 4.24. An arbitrary instance of the CLOSEST-POINT problem involving

sets A and Q of size n , and m , respectively, can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal.

□

4.6 Stabbing-Related Problems

As with the previous section, only Stage 2 of the algorithm is presented below.

Stage 2. Begin by sorting the subset $A_{i,j}$ of line segments in each $R_{i,j}$ in decreasing order of the y -coordinate of their top and bottom endpoints. The sorting can be performed in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time using any optimal algorithm for unenhanced meshes [47, 83]. Let $e_1, e_2, \dots, e_{2s^2}$ be the resulting sequence of endpoints stored in row-major order in $R_{i,j}$. Every processor in the mesh stores exactly two endpoints. The processing of Stage 2 will be partitioned into two substages, each solving a different instance of the MQ problem.

Substage 2.1. The last column of $R_{i,j}$ contains, in top-down order, the endpoints $e_{2s-1}, e_{2s}, e_{4s-1}, e_{4s}, \dots, e_{2s^2-1}, e_{2s^2}$. Now the instance of the MQ problem that is solved in $R_{i,j}$ has the following parameters:

- the sorted sequence $E_{i,j} = e_{2s-1}, e_{2s}, e_{4s-1}, e_{4s}, \dots, e_{2s^2-1}, e_{2s^2}$ of endpoints of line segments,
- the set Q of query-lines,
- a decision problem $\psi : Q \times E_{i,j} \rightarrow \{\text{"yes"}, \text{"no"}\}$ such that for every query line q_u and for every endpoint $e \in E_{i,j}$ $\psi(q_u, e) = \text{"yes"}$ if and only if endpoint e is above q_u , and
- for every query q_u , let S_u be the set of endpoints e in $E_{i,j}$ for which $\psi(q_u, e) = \text{"yes"}$.

Define $f(S_u) = \max\{k \mid e_k \in S_u\}$ (i.e., the endpoint with the least y -coordinate that is above q_u).

For every query q_u , $f(S_u)$ referred to as the *row rank* of q_u in $R_{i,j}$. To solve the above instance of the MQ problem, replicate the last column of $R_{i,j}$ throughout the submesh. This is done, in the obvious way, in $O(s) = O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time.

Next, in each of the leftmost $\frac{m}{s}$ columns of $R_{i,j}$, the items received are moved vertically, in lock step, in such a way that in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time every query-line meets every endpoint. As a consequence, every query has enough information to compute its row rank. Thus, the following result is obtained.

Lemma 4.25. The row ranks of the queries in every $R_{i,j}$ can be determined in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. \square

Substage 2.2. The goal of this stage is to solve yet another instance of the MQ problem in each submesh $R_{i,j}$. More specifically, this instance involves the following parameters:

- the sorted sequence $A_{i,j} = d_1, d_2, \dots, d_{s^2}$ of line segments,
- the set Q of queries,
- a decision problem $\phi : Q \times A_{i,j} \rightarrow \{\text{"yes"}, \text{"no"}\}$ such that for every $1 \leq u \leq m$ and $d \in A_{i,j}$, $\phi(q_u, d) = \text{"yes"}$ if and only if query-line q_u intersects the line segment d , and
- for every u ($1 \leq u \leq m$), let S_u be the set of line segments d in $A_{i,j}$ for which $\phi(q_u, d) = \text{"yes"}$. Define $f(S_u) = |S_u|$, that is, the number of segments in $R_{i,j}$ stabbed by the query-line q_u .

The processing in Substage 2.2 is motivated by the following simple observation whose proof is immediate.

Observation 4.26. Let q_u be a query-line specified by its equation $q_u = y_u$. The number of line segments in $R_{i,j}$ stabbed by q_u is precisely the number of line segments whose top endpoint has a higher y -coordinate than y_u and whose bottom endpoint has a lower y -coordinate than y_u . \square

Consider again the sorted sequence $e_1, e_2, \dots, e_{2s^2}$ and assign each top endpoint in this sequence a weight of $+1$ and to each bottom endpoint a weight of -1 . Perform a prefix sum on the resulting weighted sequence. For every endpoint e of a line segment in $R_{i,j}$ the resulting value of the prefix sum is exactly the number of segments intersected by a horizontal line through e . Now both sorting and prefix sum computation is performed in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time.

Next, step is to identify for every query q_u the unique pair e_p, e_{p+1} of endpoints with the property that $e_p > y_u > e_{p+1}$. Once this is done, the desired solution of q_u is the value of the previous prefix sum for e_p . The task of identifying the pair e_p, e_{p+1} is done as follows. Using an optimal sorting algorithm for meshes [47, 83], the sequence of queries-lines in each submesh $R_{i,j}$ is sorted in row-major order by row rank in the first $\frac{m}{s}$ rows of the submesh. Each of the first $\frac{m}{s}$ rows of $R_{i,j}$ will be termed a *query-row*. Recall that every processor in the first $\frac{m}{s}$ rows of $R_{i,j}$ contains two endpoints and one query-line. A query-row of $R_{i,j}$ is called *pure* if all the queries in the row share the same row rank. Otherwise, the query-row is termed *impure*. From here, the computation proceeds in two stages. In the first stage one finds the solution to queries in pure query rows in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time; in the second, one finds the solution to queries in impure query-rows in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. For the full details refer to Section 4.3. In summary, the following result is stated.

Lemma 4.27. The task of computing for every query-line in Q the number of line segments in each submesh $R_{i,j}$ it intersects can be carried out in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time. \square

Consequently, the following result has been proved.

Theorem 4.28. An arbitrary instance of the MULTI-STABBING problem involving a set of n line segments and a set of m query-lines can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal. \square

The POLY-LOCATION is solved by reducing it to an instance of the MULTI-STABBING problem. Details can be found in Section 2.5 of Chapter 2. Consequently, the following result is obtained.

Theorem 4.29. An arbitrary instance of the POLY-LOCATION problem involving an n -vertex simple polygon a set of m query-points can be solved in $O(m^{\frac{1}{3}}n^{\frac{1}{6}})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. \square

CHAPTER 5

THE SORTED MATRIX ALGORITHM ON THE MMB

The purpose of this chapter is to devise time optimal solution for the Batched Sorting and Ranking problem (BSR), introduced in Chapter 3, on an MMB. Just as the solution presented in Chapter 3, the algorithm proceeds in three stages. The main difference is that the knowledge of communication subsystem (local connections and buses) and the layout of processors allows for efficient implementation of the stages. This leads to a provably time-optimal solution. Recall, that a matrix with its rows and the columns independently sorted is referred to as a sorted matrix. A matrix is said to be fully sorted if its entries are sorted in row-major (resp. column-major) order.

Throughout this chapter, a generic instance of the BSR problem involves a sorted matrix A of size $\sqrt{n} \times \sqrt{n}$ stored one item per processor in a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$ and a collection Q of m , ($1 \leq m \leq n$), queries stored in the first $\frac{m}{\sqrt{n}}$ columns[†] of the MMB. The queries are of two types: *search queries* and *rank queries*. The set Q of queries is an arbitrary mix of the two query types. To avoid handling double subscripts, the items of matrix A will be

[†]In the remainder of the chapter, for simplicity it is assumed that $\frac{m}{\sqrt{n}}$ is an integer.

enumerated, in row major order, as a_1, a_2, \dots, a_n . At this point it is appropriate to explain why the m queries lie in the leftmost $\frac{m}{\sqrt{n}}$ columns of the mesh. It is assumed that the mesh with multiple broadcasting communicates with the outside world via I/O ports placed along the leftmost column of the platform. This is consistent with the view that enhanced meshes can serve as fast coprocessors for computers. In such a scenario, the host computer passes the queries on to the enhanced mesh in batches of \sqrt{n} . Thus, in the presence of m input queries, the leftmost $\frac{m}{\sqrt{n}}$ columns will receive data.

The contribution of this chapter is twofold. Firstly, it is proved that any algorithm that solves the BSR problem must take at least $\Omega(\max\{\log n, \sqrt{m}\})$ time in the worst case. This lower bound holds for both the CREW-PRAM and for the mesh with multiple broadcasting. Secondly, a time-optimal solution to the BSR problem on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$ is provided by exhibiting an algorithm whose running time is bounded by $\Theta(\max\{\log n, \sqrt{m}\})$.

To put this contribution in perspective, it is noted that recently Bhagavathi *et al.* [13] showed that the task of solving m search or rank queries in a *fully sorted* matrix can be performed in $\Theta(\sqrt{m})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Actually, in the context of fully sorted matrices the difference between the two query types vanishes, both of them being solved, essentially, in the same way.

The situation is vastly different in sorted matrices that are not fully sorted. It has been known for some time that the structure apparent in sorted matrices is not sufficient to help convert given matrix to become fully sorted. Indeed, this counterintuitive fact rediscovered by several researchers [36, 40, 45, 78] explains why querying in sorted matrices is so much harder.

In the context of sorted matrices, search queries and rank queries are very much different, requiring a different resolution strategy. It is not surprising, therefore, that the algorithm presented in this dissertation for the BSR problem is much more complicated and sophisticated than the algorithm in [13]. In order to obtain a time-optimal algorithm for the BSR problem a novel and interesting cloning strategy for the queries is developed. Consider the following overview of the strategy. The MMB is partitioned into a number of submeshes and the given queries are cloned in each of them. Having done that, the local solution of each query is obtained in each of the submeshes. Finally, since the number of clones of each query is large – larger than the available bandwidth allows to handle – a retrieving strategy is devised whereby information is gathered only from *some* of the clones. The interesting feature of this strategy is that there always exists a relatively small subset of the clones that, when retrieved, provide for the resolution of all the queries. As a consequence, the algorithm devised in this chapter is completely different from that of [13] showing the whole potential of meshes with multiple broadcasting.

In addition, the result demonstrates that for sufficiently large m , the key factor in obtaining $\Theta(\sqrt{m})$ time performance on a mesh with multiple broadcasting is not the full sortedness of the matrix but, rather surprisingly, the fact that both rows and columns are independently sorted. In this case, the running time of the algorithm only depends on m . Moreover, for values of m smaller than, approximately, $\log^2 n$, the full sortedness of the matrix is crucial in obtaining a very fast solution to the problem.

The remainder of the chapter is organized as follows: Section 5.1 presents the lower bound arguments and Section 5.2 discusses the time-optimal algorithm for the BSR problem.

5.1 Lower Bound

The purpose of this section is to establish a non-trivial lower bound for the BSR problem on meshes with multiple broadcasting. Data sets A and Q are assumed to be distributed in the MMB as described above. The lower bound arguments rely, in part, on the following fundamental result of Cook *et al.* [24].

Proposition 5.1. [24] The time lower bound for computing the logical OR of n bits on the CREW-PRAM is $\Omega(\log n)$ no matter how many processors and memory cells are available. \square

The following result of Lin *et al.* [49] is also important.

Proposition 5.2. Any computation that takes $O(t(n))$ computational steps on an n -processor mesh with multiple broadcasting can be performed in $O(t(n))$ computational steps on an n -processor CREW-PRAM. \square

It is important to note that Proposition 5.2 guarantees that if $T_M(n)$ is the execution time of an algorithm for solving a given problem on an n -processor mesh with multiple broadcasting, then there exists a CREW-PRAM algorithm to solve the same problem in $T_P(n) = T_M(n)$ time using n processors and $O(n)$ extra memory. In other words, “too fast” an algorithm on the mesh with multiple broadcasting implies “too fast” an algorithm for the CREW-PRAM. This observation is exploited in [49] to transfer known computational lower bounds for the CREW-PRAM to the mesh with multiple broadcasting.

It will be shown that even solving a *single* query of either the search or rank type takes $\Omega(\log n)$ time. This result will be proved for the CREW-PRAM and then ported to the mesh with multiple broadcasting by Proposition 5.2.

This is done by reducing the OR problem to the problem of solving a search query q . For this purpose, let $b_1, b_2, \dots, b_{\sqrt{n}}$ be an arbitrary input to OR. Con-

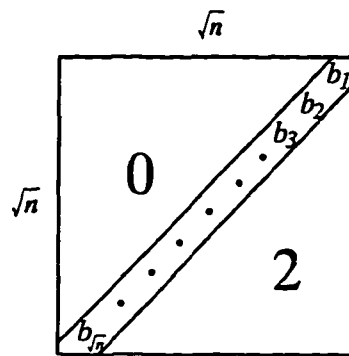


Figure 5.1: *Lower bound for solving a single query*

construct a sorted matrix A , as illustrated in Figure 5.1, by placing $b_1, b_2, \dots, b_{\sqrt{n}}$ in $A(1, \sqrt{n}), A(2, \sqrt{n}-1), \dots, A(\sqrt{n}, 1)$, and by setting for all i, j with $i+j \neq \sqrt{n}+1$:

$$A(i, j) = \begin{cases} 0 & \text{if } i+j < \sqrt{n}+1; \\ 2 & \text{if } i+j > \sqrt{n}+1. \end{cases}$$

This construction guarantees that the matrix A is sorted, regardless of the values of $b_1, b_2, \dots, b_{\sqrt{n}}$. Assign to query q the value 0.9. The answer to the OR problem is 0 if and only if the solution of the query is 0. This is because of the fact that, if the sequence $b_1, b_2, \dots, b_{\sqrt{n}}$ contains a 1, then 1 will be returned as a solution of the query, otherwise, 0 will be returned. By virtue of Proposition 5.1, any algorithm that correctly answers a search query on a sorted matrix must take $\Omega(\log \sqrt{n}) = \Omega(\log n)$ time on the CREW-PRAM, regardless of the number of processors and memory cells available.

Now to reduce the OR problem to the problem of solving a rank query q . For this purpose, let $b_1, b_2, \dots, b_{\sqrt{n}}$ be an arbitrary input to OR. Construct a sorted matrix A , as illustrated in Figure 5.1, and let the query q have the value 0.9. It should be clear that the answer to the OR problem is 0 if and only if the number of elements of A that are smaller than q is exactly $\frac{n+\sqrt{n}}{2}$. To see this, observe that by construction, every element of the matrix in the upper left triangle is strictly smaller

than the query and the only other elements that may be strictly smaller than the query lie on the diagonal, as seen from Figure 5.1. Thus, total number of elements strictly smaller than q is $\frac{n+\sqrt{n}}{2}$ if and only if all the elements on the diagonal are 0. Now Proposition 5.1 guarantees that any algorithm that correctly answers a rank query on a sorted matrix must take $\Omega(\log n)$ time on the CREW-PRAM. Combining this with Proposition 5.2, the following result is obtained.

Lemma 5.3. Any algorithm that correctly solves one search or rank query on a sorted matrix with n elements must take at least $\Omega(\log n)$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. \square

Next it will be demonstrated that every algorithm that solves the BSR problem on a *fully sorted* matrix must take $\Omega(\sqrt{m})$ time in the worst case. This will imply the same lower bound for sorted matrices. At this point, assume that A is a fully sorted matrix, and stored in the MMB as described above. The elements of A , in row major order, are referred to as a_1, a_2, \dots, a_n . In the context of fully sorted matrices, both search and rank queries are solved, essentially, the same way. Specifically, let q be an arbitrary query and let i be the subscript for which $a_i < q \leq a_{i+1}$. Clearly, if q is of rank type then its solution is i , which denotes the number of items in A strictly smaller than q . On the other hand, if q is of search type, then the item in A that is closest to q is either a_i or a_{i+1} . This observation allows us to handle, for the purpose of the lower bound, both type of queries as if they were rank queries. In turn, this implies that the following result proved in Bhavagathi *et al.* [13] can be used.

Proposition 5.4. Any algorithm that correctly solves m , $1 \leq m \leq n$, queries on a *fully sorted* matrix with n elements must take at least $\Omega(\sqrt{m})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. \square

Lemma 5.3 and Proposition 5.4 combined, provide the main result of this section.

Theorem 5.5. Any algorithm that correctly solves an instance of the BSR problem involving a sorted matrix of n items, stored one per processor, and a collection of m , ($1 \leq m \leq n$), queries, stored one per processor, in the first $\frac{m}{\sqrt{n}}$ columns of a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$ must take at least $\Omega(\max\{\log n, \sqrt{m}\})$ time. \square

5.2 A Time-Optimal BSR Algorithm

In this section it will be assumed that A is a sorted matrix and that A and the queries Q are distributed on the MMB as described in the introduction of this chapter. Similar to the discussion of Chapter 4, \mathcal{R} is viewed as consisting of submeshes $R_{i,j}$, ($1 \leq i, j \leq \sqrt{\frac{n}{m}}$), of size $\sqrt{m} \times \sqrt{m}$, and slices S_i . Figure 5.3, illustrates these subdivisions.

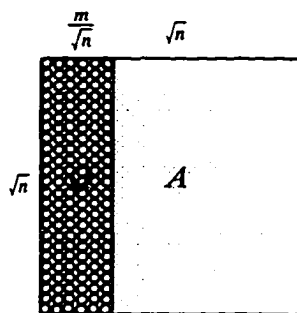


Figure 5.2: *Input to the BSR problem*

As with the other applications of the computational paradigm, the algorithm for the BSR problem proceeds in three stages which are now described with

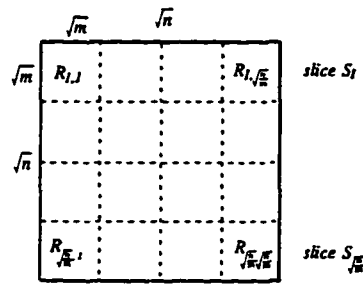


Figure 5.3: *Partition into submeshes $R_{i,j}$*

adaptation to the MMB.

Stage 1. The set Q of queries is replicated in each submesh $R_{i,j}$, creating local instances of the BSR problem.

Stage 2. In each submesh $R_{i,j}$, in parallel, the solution of the local instance of the BSR problem, is determined.

Stage 3. The solutions of the local instances of the BSR problem obtained in Stage 2 are combined into the solution of the original BSR problem.

The remainder of this section is devoted to a detailed description of each of these stages.

Stage 1.

The processing here is similar to the Stage 1 of the generic algorithm described in Section 4.2 of Chapter 4. The purpose of this stage is to replicate the set

Q of queries, in each submesh $R_{i,j}$, one query per processor. The plan is to move the queries in each column of \mathcal{R} into $\sqrt{\frac{n}{m}}$ columns of every $R_{i,j}$. Specifically, the queries in a generic column k , ($1 \leq k \leq \frac{m}{\sqrt{n}}$), of \mathcal{R} will be moved to columns $(k-1)\sqrt{\frac{n}{m}} + 1$ through $k\sqrt{\frac{n}{m}}$ of $R_{i,j}$.

Begin with every processor $P(r, k)$, ($1 \leq r \leq \sqrt{n}$) broadcasting the query it holds horizontally to the diagonal processor $P(r, r)$ as shown by the transition from Figure 5.4(a) to 5.4(b). In turn, processor $P(r, r)$ broadcasts the query received vertically to every processor $P(t\sqrt{m} + (r-1) \bmod \sqrt{m} + 1, r)$ for $t = 0, 1, \dots, \sqrt{\frac{n}{m}} - 1$, as shown in transition from Figure 5.4(b) to 5.4(c).

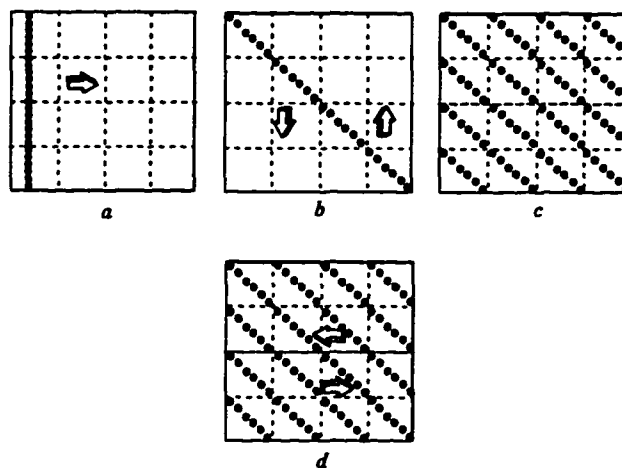


Figure 5.4: *The data movement of Stage 1*

As noted before, as a result of this data movement, the queries originally stored in column k of \mathcal{R} have been replicated in the diagonal processors of the submeshes in every slice. From now on, all slices are processed in parallel. Specifically, the queries stored by the diagonal processors of $R_{i,1}$ are replicated, using the row buses in slice S_i , into the $((k-1)\sqrt{\frac{n}{m}} + 1)$ -th column of every $R_{i,j}$ within the slice.

Next, the queries stored by the diagonal processors in $R_{i,2}$ are replicated, using the row buses in slice S_i , into the $((k-1)\sqrt{\frac{n}{m}}+2)$ -th column of each $R_{i,j}$. This process continues for each submesh in the slice. Figure 5.4(d), illustrates such a replication.

The task of replicating the \sqrt{n} queries originally stored in one column of \mathcal{R} takes $O(\sqrt{\frac{n}{m}})$ time. Therefore, as long as $m \geq \sqrt{n}$, the queries initially stored in the leftmost $\frac{m}{\sqrt{n}}$ columns of \mathcal{R} can be replicated in time $O(\sqrt{\frac{n}{m}} * \frac{m}{\sqrt{n}}) = O(\frac{m}{\sqrt{m}}) = O(\sqrt{m})$. In case $m < \sqrt{n}$ the queries are replicated in a way similar to the one described. The complexity of the data movement is, again, $O(\sqrt{m})$. With this, the goal of Stage 1 has been achieved: the queries have been replicated into each of the submeshes $R_{i,j}$, and the following result is obtained.

Lemma 5.6. The set Q of queries initially stored in the first $\frac{m}{\sqrt{n}}$ columns of \mathcal{R} can be replicated, one query per processor, in each submesh $R_{i,j}$ in $O(\sqrt{m})$ time. \square

The same techniques used in Stage 2 and Stage 3 of the BSR algorithm for the ACM are employed to solve the problem here (Section 3.1, Chapter 3). Therefore, to avoid repetition and maintain clarity, the main results and a brief sketch of the processing are presented with all the proofs omitted.

Stage 2.

At the end of the Stage 1, having replicated the set Q of queries in each submesh, $R_{i,j}$, the original instance of the BSR problem is partitioned into several instances, each local to an $R_{i,j}$. Each local instance involves the subset of \mathcal{A} stored by the processors in $R_{i,j}$ and the entire set Q of queries.

The main goal of this stage is to solve the local instance of BSR in each submesh $R_{i,j}$. To avoid broadcasting conflicts in Stage 2, the bus system is ignored so that every submesh $R_{i,j}$ will act as an unenhanced mesh. Begin by sorting the items and queries in each $R_{i,j}$ in row-major order using an optimal sorting algorithm

for meshes [47, 59]. In the sorting process ties are broken in favor of queries. In other words, if a query and an item are equal, then in the sorted version the query precedes the item.

Let $C_{i,j} = c_1, c_2, \dots, c_{2m-1}, c_{2m}$ be the resulting sorted sequence stored, two items per processor in the submesh $R_{i,j}$. The following two results will justify the approach to solving the local instances of the BSR problem.

Lemma 5.7. Let q_k be a query of rank type and assume that $c_t = q_k$, in other words, q_k occurs in position t in the sorted sequence $C_{i,j}$. The number of items in $R_{i,j}$ strictly smaller than q_k equals the number of items preceding q_k in $C_{i,j}$. \square

For solving all rank type queries in $R_{i,j}$, Lemma 5.7 motivates the strategy of assigning weights w_t for each c_t as dictated by equation 3.1. Next, compute the prefix sums of the sequence $c_1, c_2, \dots, c_{2m-1}, c_{2m}$ using the weights assigned in (3.1) and let $e_1, e_2, \dots, e_{2m-1}, e_{2m}$ be the result. By virtue of Lemma 5.7, the value e_t corresponding to $c_t = q_k$ is exactly the number of items in $R_{i,j}$ strictly smaller than q_k . Therefore, all the rank queries can be solved in the time of sorting and of prefix sums computation which is $O(\sqrt{m})$ [59, 60].

The task of handling search queries requires a different approach. This again involves assigning weights to the sorted sequence $C_{i,j} = c_1, c_2, \dots, c_{2m-1}, c_{2m}$. Referring to Figure 3.4,

Lemma 5.8. For all the search queries in some sequence s_p the solution is either $c_{\alpha-1}$ or $c_{\beta+1}$. \square

Lemma 5.8, along with weight allocation strategy described by equations 3.2 and 3.3 lead to the following result.

Lemma 5.9. The task of solving the local instance of the BSR problem in each

submesh $R_{i,j}$ can be performed, in parallel, in $O(\sqrt{m})$ time. \square

Stage 3.

At the end of Stage 2, each processor of a generic submesh $R_{i,j}$ stores, along with query q_k , its local solution $\sigma(i, j, k)$. In case q_k is a search query $\sigma(i, j, k)$ denotes the item in A closest to q_k ; in case q_k is a rank query $\sigma(i, j, k)$ denotes the number of items in A that are strictly smaller than q_k . The goal of Stage 3 is to combine these local solutions into the solution of q_k in the original instance of the BSR problem.

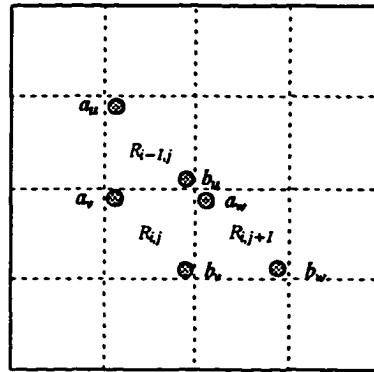


Figure 5.5: *Illustrating the proof of Lemma 5.10.*

In preparation for this, the first task of this stage is to arrange, in every submesh $R_{i,j}$, the ordered pairs $(q_k, \sigma(i, j, k))$ in row-major order, sorted by subscript k . Recall that using an optimal sorting algorithm for meshes [59], this task can be performed in $O(\sqrt{m})$ time. Note that, after sorting, the tuple $(q_k, \sigma(i, j, k))$ occupies the same relative position in each of the submeshes $R_{i,j}$.

From now on, the processing relies on the technical property of submeshes being *critical* with respect to queries. This is similar to the property of processors

being critical with respect to queries discussed in Chapter 3. Referring to Figure 5.5, a submesh $R_{i,j}$ is said to be *critical* with respect to a query q_k if q_k is larger than the entry a_v in the northwest corner of $R_{i,j}$ but not greater than the entry b_v in the southeast corner of $R_{i,j}$. The following result is key in deriving a time-optimal algorithm for the BSR problem. Refer to Figure 5.5. \square

Lemma 5.10. If a submesh $R_{i,j}$ is critical with respect to a query q_k , then at most one of the submeshes $R_{i-1,j}$ and $R_{i,j+1}$ may be critical with respect to q_k . \square

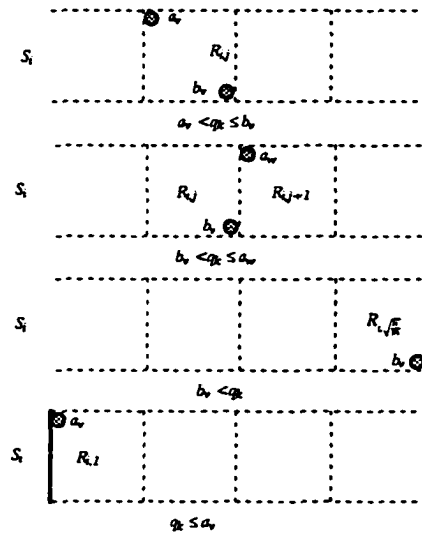


Figure 5.6: The concept of active copy of query q_k , for an MMB

Consider a generic slice S_i . For further reference, a copy of query q_k in some submesh $R_{i,j}$ is called *active* if one of the conditions (a1)–(a4) below is satisfied. Refer to Figure 5.6 for an illustration.

- (a1) $R_{i,j}$ is critical with respect to query q_k .
- (a2) Slice S_i contains no critical submesh with respect to query q_k and, for some $j < \sqrt{\frac{n}{m}}$, q_k is larger than all items in $R_{i,j}$ but smaller than or equal to all

items in $R_{i,j+1}$.

- (a3) Query q_k is larger than all the items in slice S_i ; in this case the copy of q_k in $R_{i,\sqrt{\frac{n}{m}}}$ is active.
- (a4) Query q_k is smaller than or equal to all the items in slice S_i ; in this case the copy of q_k in $R_{i,1}$ is active.

The leftmost submesh of a slice containing an active copy of a query q_k will be referred to as *leading* with respect to q_k . All the above information is computed in the following way. For determining what submeshes $R_{i,j}$ are critical with respect to a given query, the only information needed are the values in the northwest and southeast corners of the submesh. In $O(\sqrt{m})$ time these values can be circulated within the submesh and every processor becomes aware of them. Next, every processor in $R_{i,j}$ needs to be informed about the values of the items in the northwest and southeast corners of the neighboring submeshes in its own slice. Again, this information can be obtained in $O(\sqrt{m})$ time in the obvious way. With this information available, critical submeshes and active copies of all queries can be found in constant time.

The strategy for combining the solutions of queries in every $R_{i,j}$ into the global solution involves a considerable amount of data movement. To restrict the running time to $\Theta(\sqrt{m})$ the buses will be used for the data movement. Lemma 5.10 motivates the assignment of buses to *active* copies of queries according to the following rules, illustrated in Figure 5.7.

- (r1) The copy of q_k that belongs to the leading submesh in slice S_i is assigned the horizontal bus in its own row.

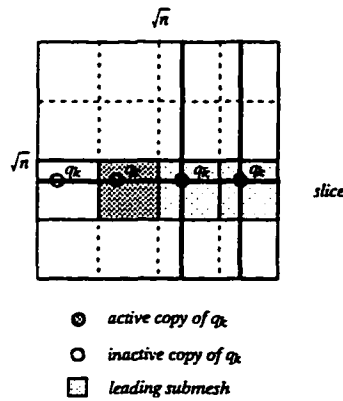


Figure 5.7: Assignment of buses

(r2) All the remaining active copies of q_k in S_i are assigned the vertical bus in their own column.

The following result shows that rules (r1) and (r2) lead to conflict-free broadcasting.

Lemma 5.11. If every active copy of q_k broadcasts simultaneously on the assigned bus, no broadcast conflict will arise.

Proof. First, no broadcast conflicts can occur on horizontal buses. To see this, note that if the horizontal bus was assigned to a copy of q_k , then either there exists only one active copy of q_k in slice S_i (in case the copy of q_k in the leading submesh is active by rules (a2)–(a4)) and so no other copy of q_k attempts to use the same bus, or else, the copy comes from a critical submesh. By rule (r2) all the other active copies in the same slice will use vertical buses and, again, no conflict can arise.

Next, to show that no conflicts can arise on vertical buses. Suppose the contrary and let i be the largest subscript for which a broadcast conflict occurs when the copy of q_k in slice S_i broadcasts vertically on its assigned bus. Without loss of generality, assume that q_k belongs to $R_{i,j+1}$. The conclusion of Lemma 5.10, along with the maximality of i imply that the copy of q_k in submesh $R_{i-1,j+1}$ is

also using the same vertical bus. This implies that neither $R_{i-1,j+1}$ nor $R_{i,j+1}$ are leading submeshes (with respect to q_k) in slice S_{i-1} , and S_i , respectively. However, now $R_{i-1,j}$, $R_{i,j}$, and $R_{i,j+1}$ must be critical with respect to q_k , contradicting Lemma 5.10 \square

It is important to note that the total number of active copies of any query q_k is at most $2\sqrt{\frac{n}{m}}$. This follows immediately from Lemma 5.11, since the assignment of buses to active copies of q_k leads to no two copies using the same bus. Since at most $\sqrt{\frac{n}{m}}$ copies of q_k are assigned horizontal buses and at most $\sqrt{\frac{n}{m}}$ copies of q_k are assigned vertical buses, the conclusion follows.

The active copies of query q_k carry enough information to yield the correct overall solution of q_k . This is due to the following result.

Lemma 5.12. Let q_k be a search query and let a be an item in A closest to q_k . There exists an active copy of q_k in some submesh $R_{i,j}$ such that either $a = \sigma(i, j, k)$ or $a = \sigma(i, j - 1, k)$ or $a = \sigma(i, j + 1, k)$. \square

Lemma 5.12 suggests an obvious way of updating the solutions of active copies of a search query q_k for which the details follow.

- If the active copy of q_k belongs to a critical submesh $R_{i,j}$ and $R_{i,j-1}$ is not critical, then the copy of q_k in $R_{i,j}$ updates its solution $\sigma(i, j, k)$ by combining it with $\sigma(i, j - 1, k)$.
- If the active copy of q_k belongs to a critical submesh $R_{i,j}$ and $R_{i,j+1}$ is not critical, then the copy of q_k in $R_{i,j}$ updates its solution $\sigma(i, j, k)$ by combining it with $\sigma(i, j + 1, k)$.
- If the copy of q_k is active because of rule (a2), then it updates its solution $\sigma(i, j, k)$ by combining it with $\sigma(i, j + 1, k)$.

Lemma 5.13. Let q_k be a rank query. The active copies of q_k in a generic slice S_i carry enough information to compute the number of items in S_i strictly smaller than q_k .

Proof. First, if all copies of q_k in slice S_i are active then the sum of their local solutions $\sigma(i, j, k)$ is exactly the number of items in S_i strictly smaller than q_k .

Assume, therefore, that not all copies of q_k in slice S_i are active. Consider the active copy of q_k in the *leading submesh* of S_i with respect to q_k .

- If this copy is active by rule (a4) then its solution $\sigma(i, j, k)$ must be 0, which is the correct number of items in S_i strictly smaller than q_k .
- If this copy is active by rule (a3) then its solution $\sigma(i, j, k)$ is updated to read $m\sqrt{n}$, which is the correct number of items in S_i strictly smaller than q_k .
- If this copy is active by rule (a1) or (a2) then its solution $\sigma(i, j, k)$ is updated to read $\sigma(i, j, k) + (j - 1)m$, which is the correct number of items in S_i strictly smaller than q_k in all submeshes $R_{i,1}, R_{i,2}, \dots, R_{i,j}$.

It is important to note that the solutions of the other active copies of q_k are *not changed* by the updates. Thus, after the required updates, the collection of active copies of q_k in slice S_i carry enough information to correctly compute the number of items in S_i smaller than q_k . The conclusion follows. \square

The next task of Stage 3 is to move all the active copies of queries to diagonal submeshes $R_{i,i}$, $1 \leq i \leq \sqrt{\frac{n}{m}}$, as illustrated in Figure 5.8. This task can be performed in two broadcast rounds as follows. The first round proceeds row by row in each submesh $R_{i,j}$. Specifically, all active copies of the queries in the first row of the $R_{i,j}$'s that have been assigned vertical buses broadcast their local solution on this bus to the corresponding processor in the diagonal submesh $R_{j,j}$. Following this, all the

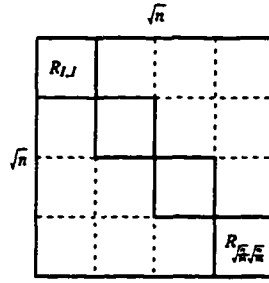


Figure 5.8: *Target of the data movement in Stage 3, for an MMB*

queries in the second row will broadcast vertically. This continues for each row of the submesh. By Lemma 5.11 no broadcast conflicts will arise. Since every $R_{i,j}$ has \sqrt{m} rows, this first round takes $O(\sqrt{m})$ time.

The second broadcast round involves broadcasting along horizontal buses. This time, the columns of every $R_{i,j}$ are handled one by one. Since there are \sqrt{m} columns in each submesh, the second round takes $O(\sqrt{m})$ time. As illustrated in Figure 5.9, it is possible for two active copies of the same query q_k to be sent to the same location of a diagonal submesh $R_{j,j}$, one copy via a horizontal bus and one via a vertical bus. By Lemma 5.11, the number of such copies is restricted to at most two. Furthermore, one of them will arrive in one broadcast round (on vertical buses) while the second will arrive on horizontal buses. The processor receiving them will proceed to combine the respective solutions. In summary, the following result is stated.

Lemma 5.14. The solutions of all active copies of queries in Q can be broadcast to the diagonal submeshes $R_{i,i}$ one per processor in $O(\sqrt{m})$ time. \square

To complete the algorithm, the various copies of queries in Q moved to the diagonal submeshes will be collected and combined. The idea is to move all the active copies of the same query from the diagonal submeshes $R_{i,i}$ to one or several

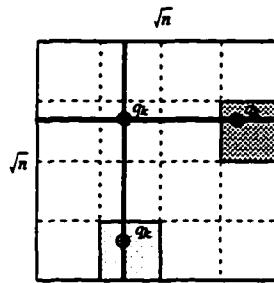


Figure 5.9: *Combining solutions, for an MMB*

adjacent rows of the original mesh. Specifically, in case

$$\sqrt{n} \leq m \quad (5.1)$$

all the copies of the first $\frac{m}{\sqrt{n}}$ queries, $q_1, q_2, \dots, q_{\frac{m}{\sqrt{n}}}$ will be moved to the first row of the mesh, the copies of the second group of $\frac{m}{\sqrt{n}}$ queries, namely, $q_{\frac{m}{\sqrt{n}}+1}, q_{\frac{m}{\sqrt{n}}+2}, \dots, q_{2\frac{m}{\sqrt{n}}}$ will be moved to the second row of the mesh, and so on.

On the other hand, in case

$$m < \sqrt{n} \quad (5.2)$$

the copies of q_1 will be moved to the first $\frac{\sqrt{n}}{m}$ rows of the mesh, \sqrt{m} per row. The data movement for both cases is similar and will be discussed next. In preparation for this data movement the following terminology needs to be introduced.

Consider a generic copy of query q_k . The quantities associated with q_k are $r(q_k)$ and $c(q_k)$ referred to as the r -value and c -value of q_k . Here, $r(q_k)$ is the identity of the row of the mesh to which this copy will have to navigate; $c(q_k)$ is the relative position of this query among the copies moved to row $r(q_k)$. The processor storing q_k can compute $r(q_k)$ and $c(q_k)$ in $O(1)$ time.

At the beginning of Stage 3, in every submesh $R_{i,j}$ the queries were sorted

in row-major order. Thus, their solutions will be stored in the diagonal submeshes $R_{i,i}$ in the same relative order.

Further, using vertical buses, the copies of the queries in the first row of every submesh $R_{i,i}$ are moved to the row of the mesh corresponding to their r -value. Specifically, a generic copy of query q_k stored by a processor $P(i, j)$ will be broadcast vertically to processor $P(r(q_k), j)$. It is crucial to note that the queries are also sorted in row-major order by their r -values and so no broadcasting conflicts can arise. Proceeding sequentially, all the \sqrt{m} rows of the $R_{i,i}$'s are processed as described. Thus, in $O(\sqrt{m})$ time all the copies will be broadcast to the row of the mesh corresponding to their r -value. No processor will receive more than one copy of any query in the above data movement.

From now on, the processing that takes place in each row of the mesh depends on whether (5.1) or (5.2) holds. First, assume that (5.1) is true. The processing that takes place in the first row of the mesh will be detailed, the same action being performed, in parallel in all other rows. The copies of q_1 will be broadcast to processor $P(1, 1)$ in the order of their c -values. Upon receiving the next copy of q_1 , $P(1, 1)$ combines the corresponding solutions. Since there are (at most) $\sqrt{\frac{n}{m}}$ copies of q_1 , the solution of query q_1 will be obtained in $O(\sqrt{\frac{n}{m}})$ time. The copies of the remaining queries $q_2, \dots, q_{\frac{m}{\sqrt{n}}}$ moved to row 1 will be processed similarly. Therefore, the overall time needed to solve all the queries in case (5.1) holds is bounded by $O(\sqrt{\frac{n}{m}} * \frac{m}{\sqrt{n}}) = O(\sqrt{m})$.

In case (5.2) holds, recall that the copies of a given query have been spread over $\frac{\sqrt{n}}{m}$ rows of the mesh. Again, the processing of query q_1 will be discussed, all the others being handled, in parallel, in a similar way. The $\frac{\sqrt{n}}{m}$ copies of q_1 have been moved to rows $1, 2, \dots, \frac{\sqrt{n}}{m}$, \sqrt{m} copies to each row. Now proceeding

sequentially, in order of their c -values, the \sqrt{m} copies of q_1 in each of the first $\frac{\sqrt{n}}{m}$ rows will be broadcast to the leftmost processor in these rows. These processors will be responsible for combining the solutions as described above.

By Lemmas 5.12 and 5.13, at the end of \sqrt{m} broadcasts, all the information needed to solve the queries is stored by the processors in the first column of the mesh. Specifically, processors $P(1, 1), P(2, 1), \dots, P(\frac{\sqrt{n}}{m}, 1)$ contain partial solutions corresponding to query q_1 , the next group of $\frac{\sqrt{n}}{m}$ processors contain partial solutions corresponding to query q_2 , and so on. Refer to Figure 5.10 for an illustration.

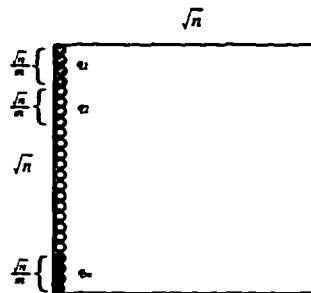


Figure 5.10: *Partial solutions contained by processors in first column*

Finally, consider diagonal submeshes D_1, D_2, \dots, D_m of size $\frac{\sqrt{n}}{m} \times \frac{\sqrt{n}}{m}$ as illustrated in Figure 5.11. For the final step of Stage 3 the diagonal submesh D_i is dedicated to solving the query q_i .

In one broadcast, the partial results stored by processors in the first column of the mesh are moved, along horizontal buses to the first column of each D_i , as depicted in Figure 5.10. Now combining the partial solutions of query q_i , ($1 \leq i \leq m$), amounts to a semigroup computation, local to D_i . Using the result of Olariu *et al.* [61] this computation can be performed in $O(\log \frac{\sqrt{n}}{m})$ time. Once the final solution of each query has been computed, it is moved back to the first column of the mesh.

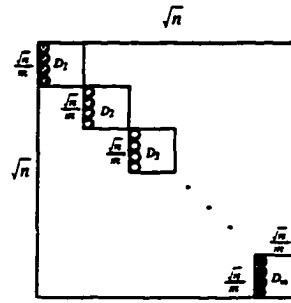


Figure 5.11: *Submeshes* D_1, D_2, \dots, D_m

Consequently, in case (5.2) holds the overall running time of the algorithm is bounded by $O(\sqrt{m} + \log \frac{\sqrt{n}}{m})$. Since for $m > 16$

$$\sqrt{m} - \log m > 0,$$

the running time of the algorithm, in case (5.2) holds, satisfies $O(\sqrt{m} + \log \frac{\sqrt{n}}{m}) \subseteq O(\sqrt{m} + \log n)$. In summary, the following result is stated.

Theorem 5.15. An arbitrary instance of the BSR problem involving a sorted matrix of size $\sqrt{n} \times \sqrt{n}$ and a set of m queries, can be solved in $O(\max\{\log n, \sqrt{m}\})$ time on a mesh with multiple broadcasting of size $\sqrt{n} \times \sqrt{n}$. Furthermore, this is time-optimal on this architecture. \square

CHAPTER 6

IMPLEMENTATION AND CONCLUSIONS

6.1 Implementation Results

To demonstrate the practical applicability of the computational paradigm, two instances of the MQ problem were implemented. These instances represent the following problems:

- the multiple point location problem, and
- the multiple rank problem.

These problems were selected because they are fundamental having a wide variety of applications. The code was written in the C programming language, using the Message Passing Interface (MPI). The code can be ported to any existing platform which supports MPI. In particular, experiments were run on the IBM SP2 and a network of workstations. IBM SP2 is multiprocessor system built using powerful RS/6000 processors. The communication medium is a multi-stage omega network.

In this implementation, the timings for the three stages of the algorithm were documented. Specifically, the three stages are referred to follows: Stage 1 is termed *the broadcast phase*, Stage 2 is termed *the compute phase*, and Stage 3 is termed *the*

reduce phase. The remainder of this section presents the details of the timings and the supporting arguments for their trends.

6.1.1 Multiple Rank Problem

Recall the multiple rank problem, given a set A of items and a set Q of queries from a totally ordered universe, for each q in Q determine the number of items in A with smaller values. The three stages have the following following theoretical timings, with $m \leq M$:

- Broadcast phase: $T_B(m, p)$,
- Compute phase: $O((m + \frac{n}{p}) \log \frac{n}{p})$, and
- Reduce phase: $T_R(m, p)$.

As p increases, the running time of compute phase should drop and there should be an increase in the running times for broadcast and reduce phases. The increase for the $T_B(m, p)$ should be relatively lesser than that of $T_R(m, p)$. This is because of the fact that, the total number of elements involved in Stage 3 (reduce phase) is mp . This implies that both the communication as well as the computation will increase for the reduce phase.

This problem was run using up to 32 processors with randomly generated values for items and queries. Figure 6.1 indicates the running times for $n = 8000000$, $m = 1000000$. From this figure it is clear that the general trends for compute phase and the reduce phase are decrease and increase in the times, respectively which is expected. Experiments were performed with other input values, with similar results. The best case speedup obtained was 26 with thirty two processors (here $n = 7500000$, $m = 100000$) as illustrated in Figure 6.2.

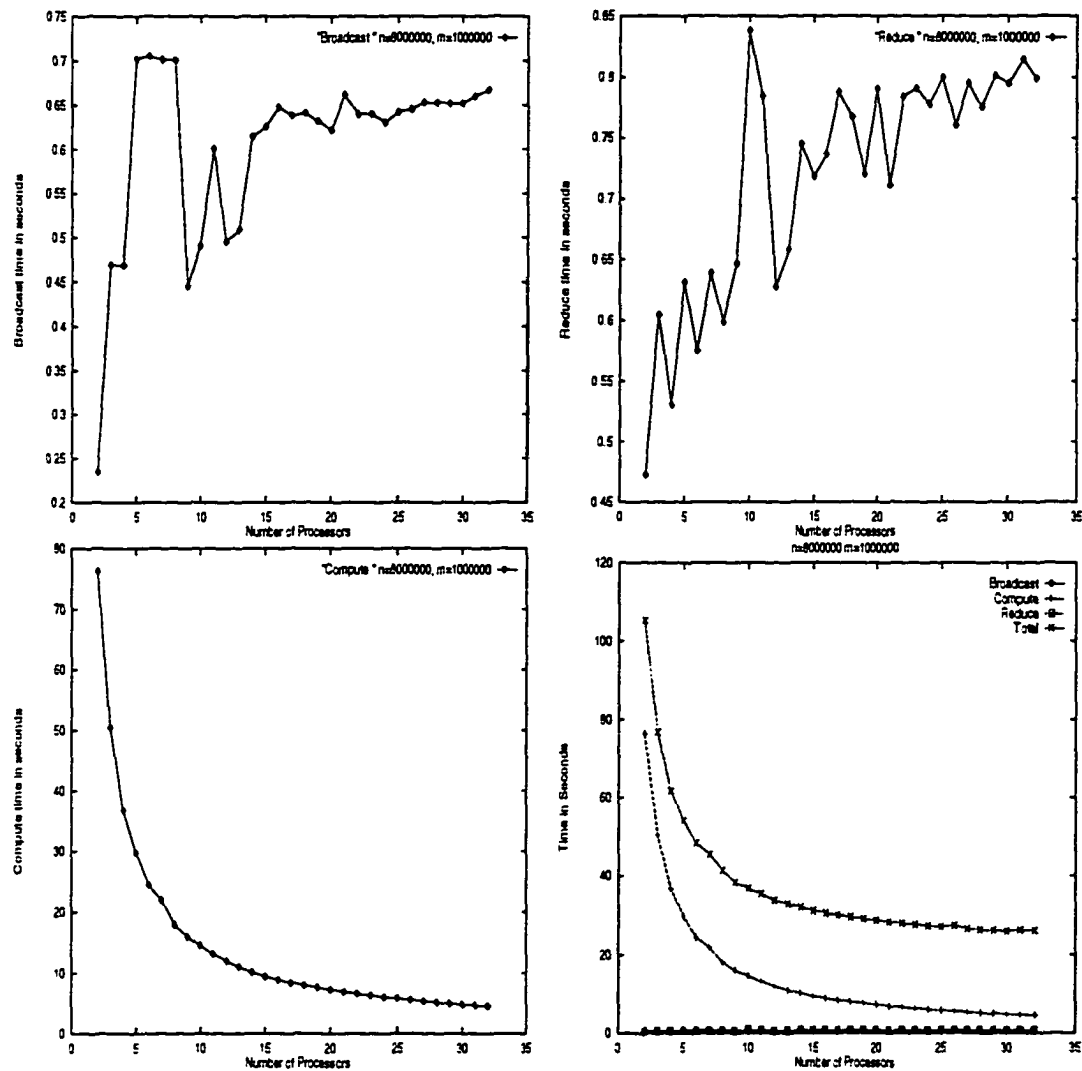


Figure 6.1: *Running times for multiple rank problem*

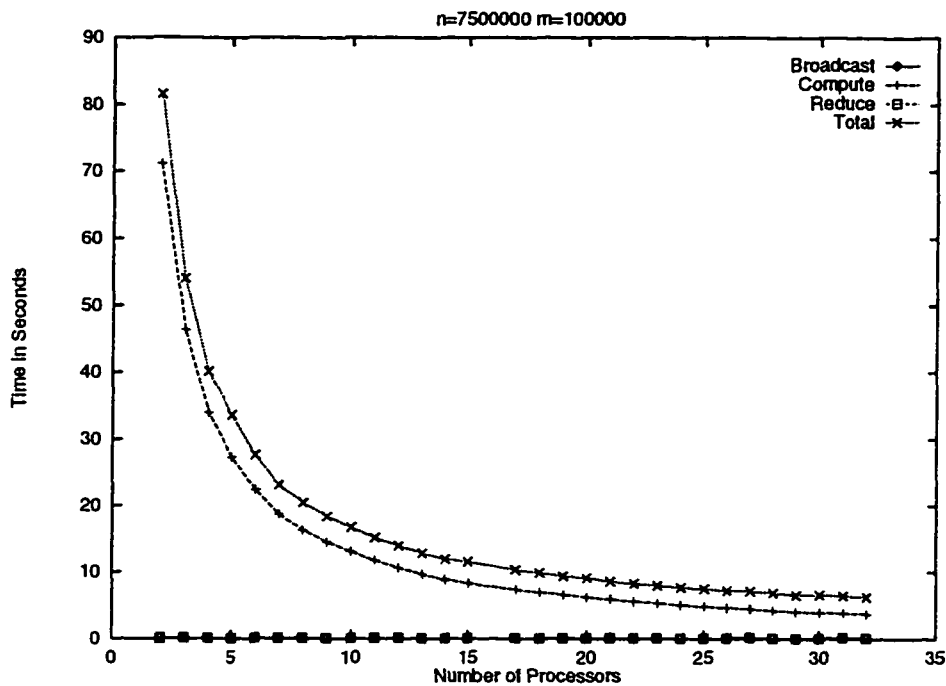


Figure 6.2: *The best case running times for multiple rank problem*

6.1.2 Multiple Point Location Problem

Recall the multiple point location problem, given sets A and Q of points in the plane, determine for each query q whether it lies within the $\text{CH}(A)$. If not, return the tangents from q to $\text{CH}(A)$. In the implementation of this problem, for the local convex hull computations Graham scan [67] was used. The three stages have the following theoretical timings:

- Broadcast phase: $T_B(m, p)$,
- Compute phase: $O(\frac{n}{p} \log \frac{n}{p} + m \log h)$,
- Reduce phase: $T_R(m, p)$.

In the compute phase h denotes the number of points on the local convex hull. This motivates for four kinds of inputs to the problem:

1. random input for items and queries,
2. random input for items and all queries outside the $CH(A)$,
3. all the items on a circle and all queries inside the $CH(A)$, and
4. all the items on a circle and all queries outside the $CH(A)$.

In this implementation the variations in the running times for these four kinds of inputs was not very significant. Although, there is a noticeable difference between cases 1 and 4. The best speed up obtained was 16.79 with 32 processors. This was obtained for $n = 4000000$, $m = 600000$, and the nature of input was case 1, refer to Figure 6.3. Again, experiments were performed for different input sizes with similar results.

Figure 6.3 represents case 1, that is, items and queries were generated randomly. Similarly, Figures 6.4, 6.5, and 6.6 represent cases 2, 3, and 4, respectively. Figure 6.7, indicates the running times on a network of workstations. The results are for case 4, here the computation time decreases and the reduce time increases.

Even for the multiple point location problem, the graphs show expected trends. Except for the reduce phase, where for the first 5 processors the running time drops instead of increasing. It would be interesting to see how the compute phase performs if the graham scan, whose running time is $n \log n$, is replaced with the Jarvis' march algorithm whose running time is nh , where h denotes the number of points on the hull and n being the input size, refer to [67] for a detailed discussion of these algorithms.

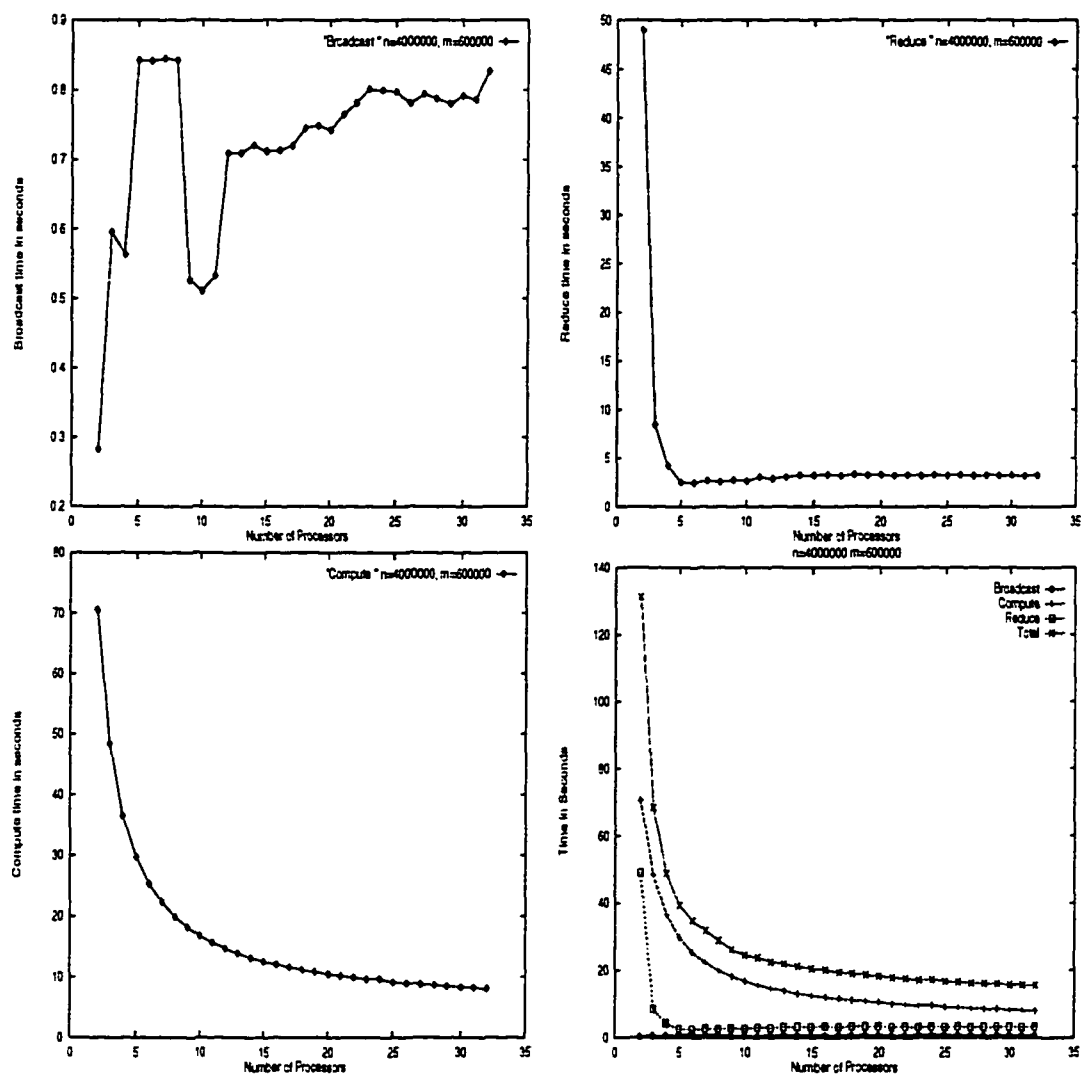


Figure 6.3: *Multiple point location problem: case 1*

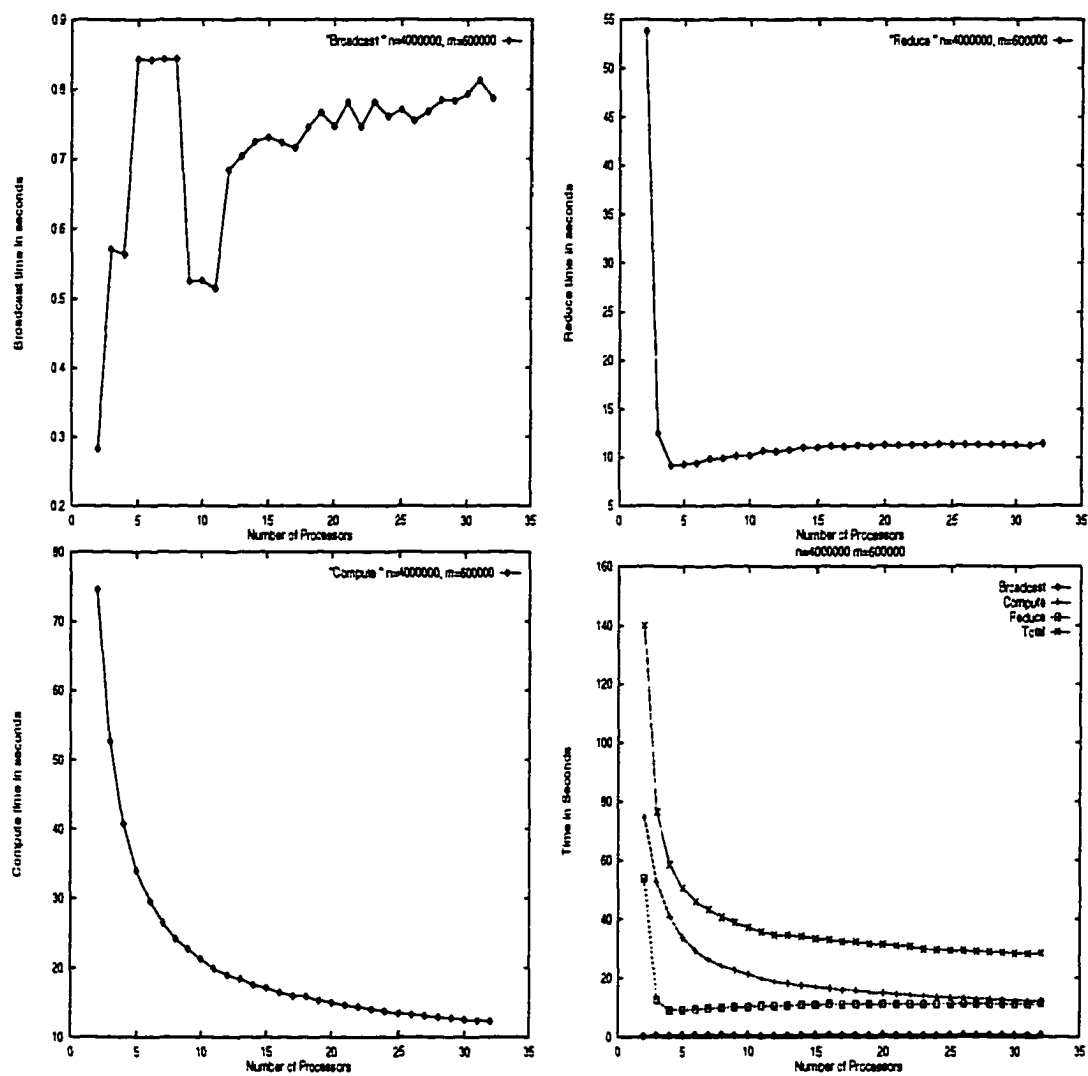


Figure 6.4: *Multiple point location problem: case 2*

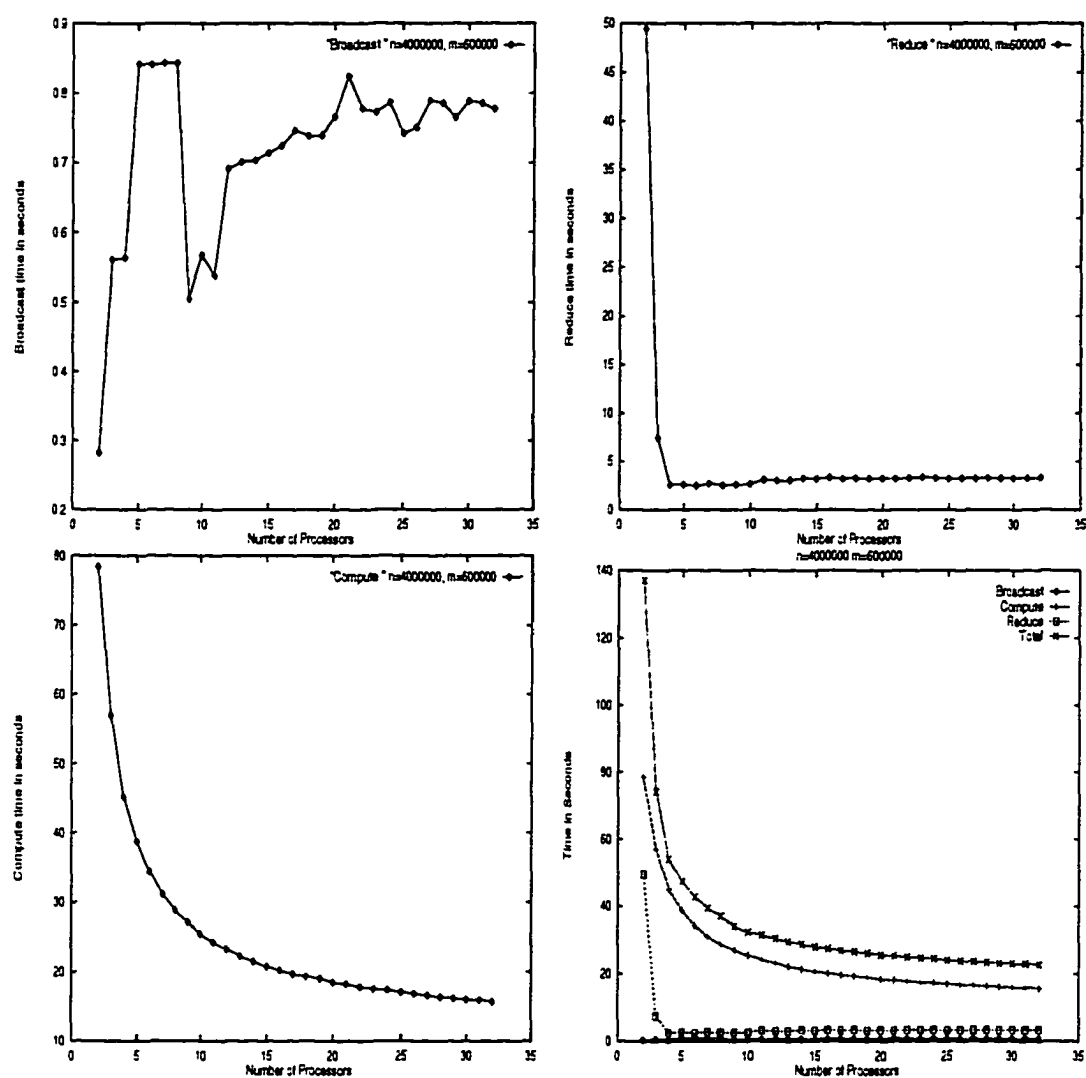


Figure 6.5: Multiple point location problem: case 3

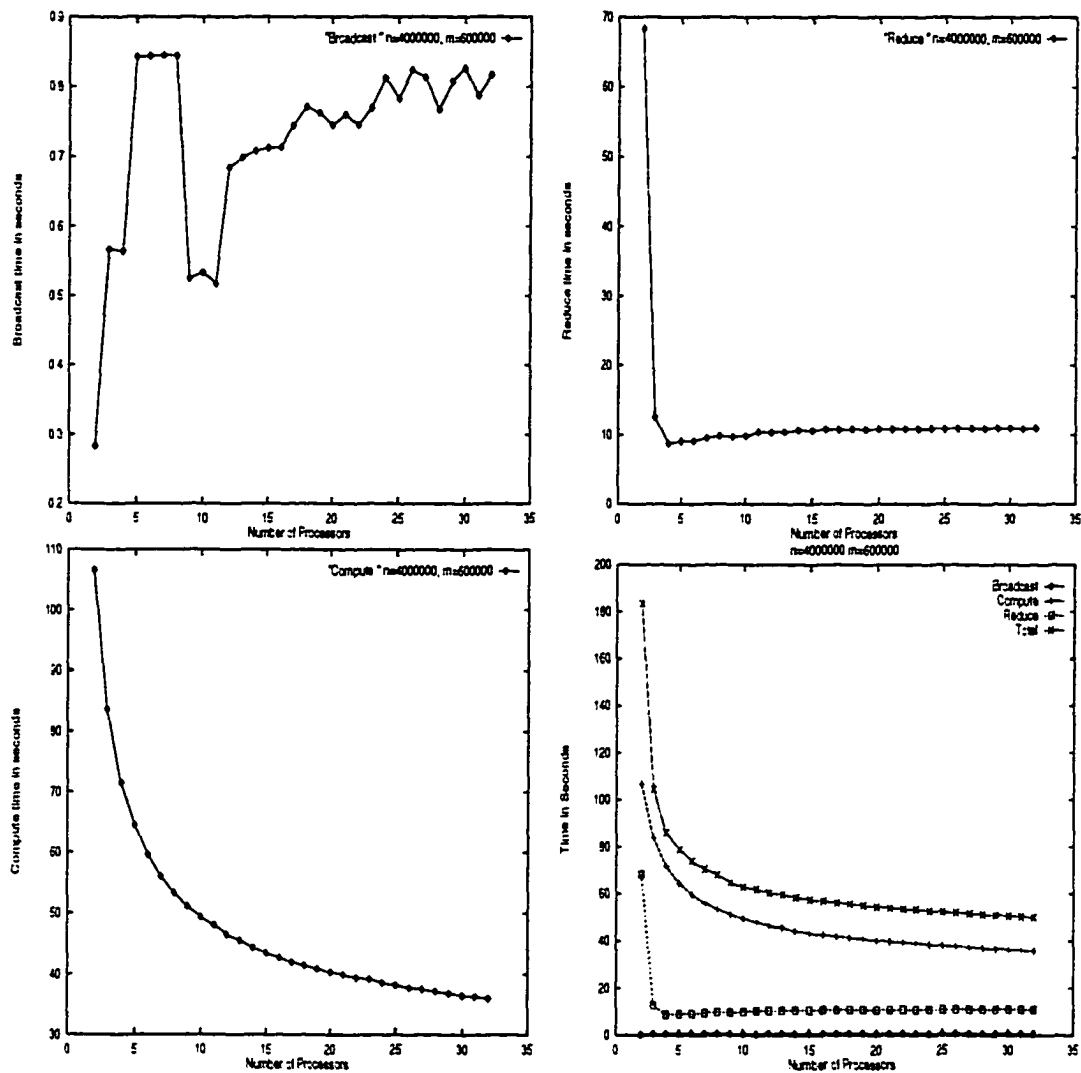


Figure 6.6: *Multiple point location problem: case 4*

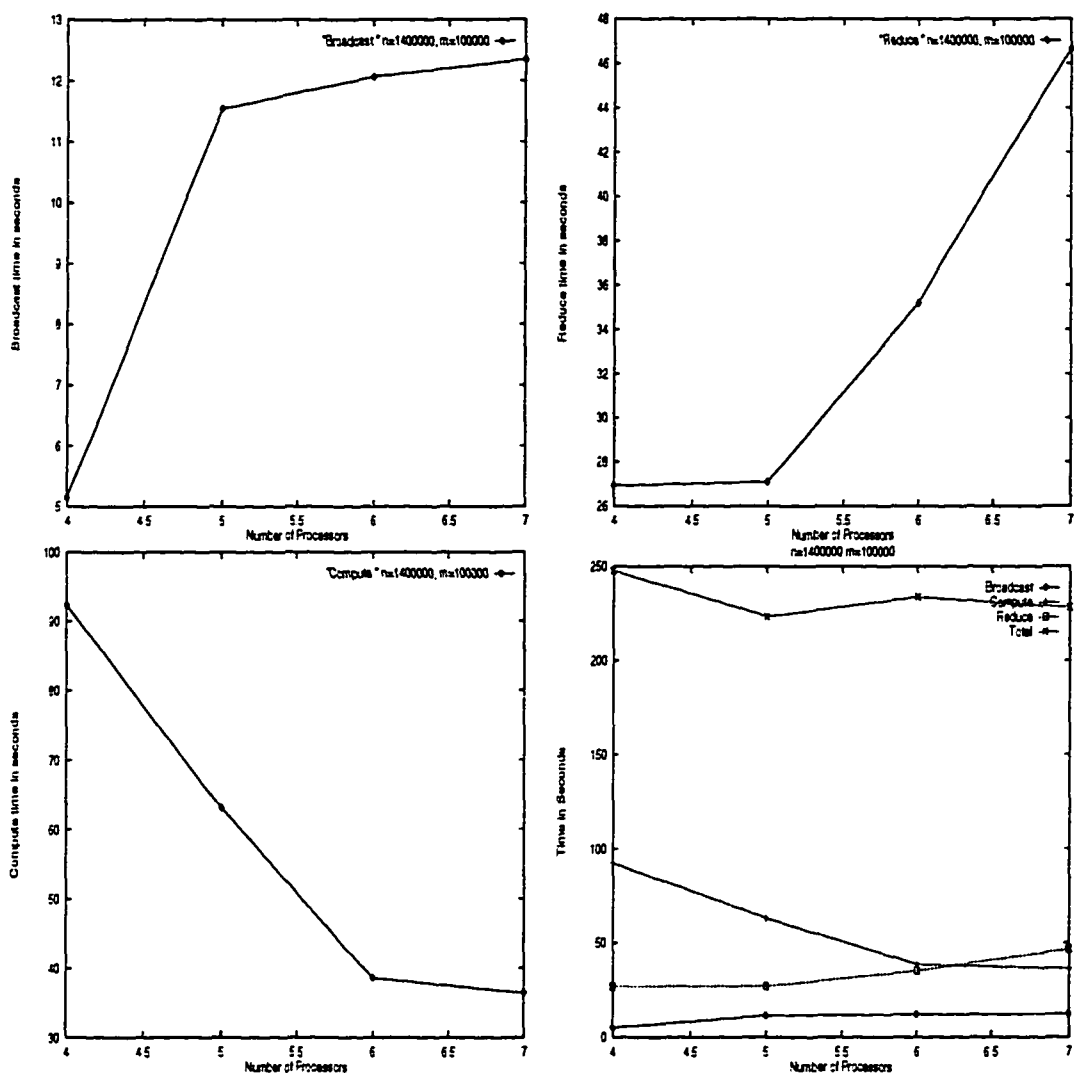


Figure 6.7: *Running times for a network of workstations*

6.2 Conclusions and Future work

The main contribution of this dissertation is the computational paradigm presented in the form a unifying algorithmic framework (the MQ problem) and a generic solution of the MQ problem. The power of the paradigm was demonstrated by identifying and providing solutions for fundamental problems in diverse areas of computer science. This was accomplished by providing a generic solution for the MQ problem, proving that each of these problems are special instances of the MQ problem, and solving these problems individually. In particular, the following problems were solved.

- Rank Related
 - Multiple Rank
 - Histogram
- Proximity Related
 - Multiple Point Location
 - Convex Hull Inclusion
 - Separability
 - Multiple Closest Point
- Segment Related
 - Multiple Closest Segment
 - Multiple Rectangle
 - Multiple Circle
- Stabbing Related
 - Multiple Stabbing

– Point Location in Simple Polygon

A major contribution is the presentation of time-optimal solutions for some of these problems on an MMB. Another major contribution of this dissertation is the Batched Searching and Ranking in *sorted matrices*. The BSR problem was also demonstrated to be an instance of the MQ problem. Here the main difference is the structure that sorted matrices offer. It was demonstrated that time-optimal solutions can be obtained by coupling the paradigm with sorted matrices on an MMB.

In this work, two fundamental problems namely, the multiple point location, and the multiple rank, were implemented on an IBM SP2 and a network of workstations. The results obtained indicate the expected trends.

Future work involves, trying to extend the function f , of the MQ problem, to encompass a wider variety of problems. Also an implementation of some more problem is in order, with the results tested on more platforms like the Intel Paragon.

BIBLIOGRAPHY

- [1] S. G. Akl and K. A. Lyons, *Parallel Computational Geometry*, Prentice-Hall, Englewood Cliffs, New Jersey, 1993.
- [2] S. G. Akl and J. Meijer, Parallel binary search, *IEEE Transactions on Parallel and Distributed Systems*, 1, (1990), 247–250.
- [3] S. G. Akl and G. T. Toussaint, Efficient convex hull algorithms for pattern recognition applications, *Proceedings Fourth International Joint Conference on Pattern Recognition*, (1978), 483–487.
- [4] M. J. Atallah, R. Cole, M. T. Goodrich, Cascading divide-and-conquer: a technique for designing parallel algorithms, *SIAM Journal on Computing*, 18(3), (1989), 499–532.
- [5] M. J. Atallah and M. T. Goodrich, Efficient plane sweeping in parallel (preliminary version), *Proceedings of the Second Annual ACM Symposium on Computational Geometry*, Yorktown Heights, New York, (June 1986), 216–225.
- [6] M. J. Atallah and J. J. Tsay, On the parallel-decomposability of geometric problems, *Proceedings of the fifth Annual ACM Symposium on Computational Geometry*, Saarbruchen, Germany, (June 1989), 104–113.
- [7] D. H. Ballard and C. M. Brown, *Computer Vision*, Prentice-Hall, Englewood

Cliffs, NJ, 1982.

- [8] A. Bar-Noy and D. Peleg, Square meshes are not always optimal, *IEEE Transactions on Computers*, 40, (1991), 196–204.
- [9] G.H.Barnes, R.M. Brown, M. Kato, D.J. Kuck, D.L. Slotnick, and R.A. Stokes, The ILLIAC IV computer, *IEEE Transactions on Computers*, 17, (1968), 746–757.
- [10] K. E. Batcher, Design of Massively Parallel Processor, *IEEE Transactions on Computers*, 29, (1980), 836–840.
- [11] D. Bhagavathi, V. Bokka, H. Gurla, S. Olariu, J. L. Schwing, and I. Stojmenovic, Time-optimal visibility-related problems on meshes with multiple broadcasting, *IEEE Transactions on Parallel and Distributed Systems*, to appear, (1995).
- [12] D. Bhagavathi, P. J. Looges, S. Olariu, J. L. Schwing, and J. Zhang, A fast selection algorithm on meshes with multiple broadcasting, *IEEE Transactions on Parallel and Distributed Systems*, 5, (1994), 772–778.
- [13] D. Bhagavathi, S. Olariu, W. Shen, and L. Wilson, A time-optimal multiple search algorithm on enhanced meshes, with applications, *Journal of Parallel and Distributed Computing*, 22, (1994), 113–120.
- [14] D. Bhagavathi, S. Olariu, J. L. Schwing, and J. Zhang, Convex polygon problems on meshes with multiple broadcasting, *Parallel Processing Letters*, 2, (1992), 249–256.
- [15] D. Bhagavathi, S. Olariu, W. Shen, and L. Wilson, A unifying look at semigroup computations on meshes with multiple broadcasting, *Proceedings of Parallel*

- Architectures and Languages Europe*, München, Germany, (June 1993), LNCS 694, 561–569.
- [16] D. Bhagavathi, V. Bokka, H. Gurla, S. Olariu, J. L. Schwing, and Zhang, Square meshes are not optimal for convex hull computation, *Proceedings of International Conference on Parallel Processing*, St-Charles, Illinois, (August 1993), III, 307–311.
 - [17] S. H. Bokhari, Finding maximum on an array processor with a global bus, *IEEE Transactions on Computers*, 33, (1984), 133–139.
 - [18] C. Chao, W. Chen, G. Chen, Multiple search problem on reconfigurable mesh, *Preprint*, Department of Computer Science and Information Engineering, National Taiwan University, Taipei, Taiwan, (1993).
 - [19] Y. C. Chen, W. T. Chen, G.-H. Chen and J. P. Sheu, Designing efficient parallel algorithms on mesh connected computers with multiple broadcasting, *IEEE Transactions on Parallel and Distributed Systems*, 1, (1990), 241–246.
 - [20] Y. C. Chen, W. T. Chen, and G.-H. Chen, Efficient median finding and its application to two-variable linear programming on mesh-connected computers with multiple broadcasting, *Journal of Parallel and Distributed Computing*, 15, (1992), 79–84.
 - [21] B. Chazelle, Computational geometry on a systolic chip, *IEEE Transactions on Computers*, 33, (1984), 774–785.
 - [22] R. Cole and M. T. Goodrich, Optimal parallel algorithms for polygons and point-set problems (preliminary version), *Proceedings of the Fourth Annual*

- ACM Symposium on Computational Geometry*, Urbana-Champaign, Illinois, (June 1988), 201–210.
- [23] C.F. Codd, *Cellular Automata*, Academic Press, New York, 1968.
 - [24] S. A. Cook, C. Dwork, and R. Reischuk, Upper and lower time bounds for parallel random access machines without simultaneous writes, *SIAM Journal on Computing*, 15, (1986), 87–97.
 - [25] M. Cosnard, J. Dupras, and A. G. Ferreira, The complexity of searching in $X + Y$ and other multisets, *Information Processing Letters*, 34, (1990), 103–109.
 - [26] M. Cosnard and A. G. Ferreira, Parallel algorithms for searching in $X + Y$, *Proceedings of the International Conference on Parallel Processing*, St. Charles, Illinois, (August 1989), 16–19.
 - [27] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. V. Eicken, LogP: Towards a realistic model of parallel computation, *Proceedings of the Fourth ACM SIGPLAN Symposium on Principles and Practice of Parallel Processing*, (May 1993).
 - [28] F. Dehne, Solving visibility and separability problems on mesh of processors, *The Visual Computer*, 3, (1988), 356–370.
 - [29] R. O. Duda and P. E. Hart, *Pattern Classification and Scene Analysis*, Wiley and Sons, New York, 1973.
 - [30] C.R. Dyer and A. Rosenfeld, Parallel image processing by memory augmented cellular automata, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3, (1981), 29–41.

- [31] H. Eddlsbrunner, *An introduction to Combinatorial Geometry*, Springer-Verlag.
- [32] A. G. Ferreira, Parallel search in sorted multisets, with applications to NP-complete problems, *Technical Report 90-92*, Laboratoire de l'Informatique du Parallélisme, Ecole Normale Supérieure de Lyon, (December 1990).
- [33] H. Freeman, Computer processing of line-drawing images, *Computer Surveys*, 6, (1974), 57-97.
- [34] H. Freeman, R. Shapira, Determining the minimum-area encasing rectangle for a arbitrary closed curve, *Communications of the ACM*, 18(7), (1975), 409-413.
- [35] L. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer graphics, principles and practice*, Second Edition, Addison-Wesley, Reading, MA, 1990.
- [36] H. Freeman and G. Pieroni, Eds., *Computer architecture for spatially distributed data*, Springer-Verlag, Heidelberg, Berlin, 1985.
- [37] G. N. Frederickson and D. B. Johnson, Generalized selection and ranking: sorted matrices, *SIAM Journal on Computing*, 13, (1984), 14-30.
- [38] M. T. Goodrich, Triangulating a polygon in parallel, *Journal of Algorithms*, 10, (September 1989), 327-351.
- [39] H. Gurla, Visibility-related problems on parallel computational models, *Doctoral Dissertation*, Old Dominion University, Norfolk, Virginia, (1996).
- [40] L. H. Harper, T. H. Payne, J. E. Savage, and E. Strauss, Sorting $X + Y$, *Communications of the ACM*, 18, (1975), 347-349.
- [41] D. Hillis, *The Connection Machine*, MIT press, Cambridge, Massachusetts, 1985.

- [42] C. S. Jeong and D. T. Lee, Parallel geometric algorithms on a mesh-connected computer, *Algorithmica*, 5, (1990), 155–178.
- [43] V. P. Kumar and C. S. Raghavendra, Array processor with multiple broadcasting, *Journal of Parallel and Distributed Computing*, 2, (1987), 173–190.
- [44] V. K. Prasanna and D. I. Reisis, Image computations on meshes with multiple broadcast, *IEEE Transactions Pattern Analysis and Machine Intelligence*, 11, (1989), 119–125.
- [45] J. L. Lambert, Sorting $X + Y$ in $O(n^2)$ comparisons, *Theoretical Computer Science*, 103, (1992), 137–141.
- [46] J.-P. Laumond, Obstacle growing in a non-polygonal world, *Information Processing Letters*, 25, (1987), 41–50.
- [47] F. T. Leighton, Tight bounds on the complexity of parallel sorting, *IEEE Transactions on Computers*, 34, (1985), 344–354.
- [48] H. Li and M. Maresca, Polymorphic-torus network, *IEEE Transactions on Computers*, 38, (1989), 1345–1351.
- [49] R. Lin, S. Olariu, J. L. Schwing, and J. Zhang, Simulating enhanced meshes, with applications, *Parallel Processing Letters*, 3, (1993), 59–70.
- [50] T. Lozano-Perez, Spatial Planning: A Configurational Space Approach, *IEEE Transactions on Computers*, 32, (1983) 108–119.
- [51] M. Lu and P. Varman, Solving geometric proximity problems on mesh-connected computers, *Proceedings of Workshop on Computer Architecture for Pattern Analysis and Image Database Management*, (1985), 248–255.

- [52] M. Maresca and H. Li, Connection autonomy and SIMD computers: a VLSI implementation, *Journal of Parallel and Distributed Computing*, 7, (1989), 302–320.
- [53] T. H. Merrett, *Relational Information Systems*, Reston Publishing, Reston, Virginia, 1984.
- [54] R. Miller, and Q.F. Stout, Geometric algorithms for digitized pictures on a mesh-connected computer, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7, (1985), 216–228.
- [55] R. Miller, and Q.F. Stout, Mesh computer algorithms for computational geometry, *IEEE Transactions on computers*, 38, (1989), 321–340.
- [56] R. Miller, and Q.F. Stout, Mesh computer algorithms for line segments and simple polygons, *Proceedings of International Conference on Parallel Processing*, St. charles, Illinois, (August 1987), 282–285.
- [57] P. D. MacKenzie and Q. F. Stout, Asymptotically efficient hypercube algorithms for computational geometry, *Proceedings of the Third Annual Symposium on the Frontiers of Massively Parallel Computation*, College Park, Maryland, (October 1990), 8–11.
- [58] A. Mirzaian, Channel routing in VLSI, *Proceedings 16-th Annual ACM Symposium on Theory of Computing*, Washington, DC, (1984), 101–107.
- [59] D. Nassimi and S. Sahni, Bitonic sort on a mesh-connected parallel computer, *IEEE Transactions on Computers*, 27, (1979), 2–7.
- [60] D. Nassimi and S. Sahni, Data broadcasting in SIMD computers, *IEEE Transactions on Computers*, 30, (1981), 101–107.

- [61] S. Olariu, J. L. Schwing, and J. Zhang, Optimal convex hull algorithms on enhanced meshes, *BIT*, 33, (1993), 396–410.
- [62] S. Olariu, J. L. Schwing, and J. Zhang, Fast computer vision algorithms for reconfigurable meshes, *International Parallel Processing Symposium*, Beverly Hills, California, (1992), 258–262.
- [63] S. Olariu and I. Stojmenović, Time-optimal proximity problems on meshes with multiple broadcasting, *Proceedings of International Parallel Processing Symposium*, Cancun, Mexico, (April 1994), 94–101.
- [64] S. Olariu and I. Stojmenović, Time-optimal nearest-neighbor computations on enhanced meshes, *Proceedings PARLE*, Patras, Greece, (July 1994).
- [65] D. Parkinson, D. J. Hunt, and K. S. MacQueen, The AMT DAP 500, 33rd *IEEE Comp. Soc. International Conf.*, (February 1988), 196–199.
- [66] B. T. Preas and M. J. Lorenzetti (Eds.) *Physical Design Automation of VLSI Systems*, Benjamin/Cummings, Menlo Park, 1988.
- [67] F. P. Preparata and M. I. Shamos, *Computational Geometry, An Introduction*, Springer-Verlag, New York, Berlin, 1988.
- [68] J. H. Reif and S. Sen, Optimal randomized parallel algorithms for computational geometry, *Proceedings of International Conference on Parallel Processing*, St. Charles, Illinois, (August 1987), 270–277.
- [69] J. H. Reif and S. Sen, Randomized algorithms for binary search and load balancing on fixed connection networks with geometric applications (preliminary version), *Proceedings of the Second ACM Symposium on Parallel Algorithms and Architectures*, Crete, (July 1990), 327–337.

- [70] S. F. Reddaway, A. Wilson, and A. Horn, Fractal graphics and image compression on a SIMD processor, *Proceedings Second Symposium on Frontiers of Massively Parallel Computation*, Fairfax, Virginia, (1988), 265–274.
- [71] A. Rosenfeld, *Picture Processing by Computers*, Academic Press, New York, 1969.
- [72] J. Rothstein, Bus automata, brains, and mental models, *IEEE Transactions on Systems, Man, and Cybernetics*, 18, (1988), 522–531.
- [73] F. Rosenblatt, *Principles of Neurodynamics*, Spartan Books, New York, 1962.
- [74] H. Samet, *The Design and Analysis of Spatial Data Structures*, Addison-Wesley, Reading, MA, 1989.
- [75] J. Serra, *Image Analysis and Mathematical Morphology*, Academic Press, New York, 1982.
- [76] J. Sklansky, Measuring concavity on a rectangular mosaic, *IEEE Transactions on Computers*, 21, (1982), 181–187.
- [77] D. Sarkar and I. Stojmenović, An optimal parallel algorithm for minimum separation of two sets of points, *Technical Report TR-89-23*, Computer Science Department, University of Ottawa, Ottawa, Ontario, (July 1989).
- [78] W. Steiger and I. Streinu, A pseudo-algorithmic separation of lines from pseudo-lines, *Information Processing Letters*, 53, (1995), 295–299.
- [79] I. Stojmenović, Computational geometry on a hypercube, *Proceedings of the International Conference on Parallel Processing*, St. Charles, Illinois, (August 1988), III, 100–103.

- [80] H. S. Stone, *High-Performance Computer Architecture*, Second Edition, Addison-Wesley, Reading, MA, 1990.
- [81] R. Tamassia and J. S. Vitter, Optimal cooperative search in fractional cascaded data structures, *Proceedings of the Second ACM Symposium on Parallel Algorithms and Architectures*, Crete, (July 1990), 307–316.
- [82] H. Tamura and N. Yokoya, Image database systems: a survey, *Pattern Recognition*, 17, (1984), 29–49.
- [83] C. D. Thompson and H. T. Kung, Sorting on a mesh-connected parallel computer, *Communications of the ACM*, 20, (1977), 263–271.
- [84] L. G. Valiant, A bridging model for parallel computation, *Communications of the ACM*, 33, (1990), 103–111.
- [85] Z. Wen, Parallel Multiple Search, *Information Processing Letters*, 37, (1991), 181–186.
- [86] I. M. Yaglom and V. G. Boltyanski, *Convex Figures*, Holt, Rinehart, and Winston, New York, 1961.

VITA

Venkatavasu Bokka was born in Kuchipudi, A.P., India on September 21, 1971. He received his Bachelor of Technology in Computer Science and Engineering from Indian Institute of Technology, Delhi, India, in August 1992. He worked as a Software Engineer for Kernex India Limited, India, from September 1992 to December 1992. In January 1993, he started working on his Ph.D Degree in Computer Science at Old Dominion University, Virginia.

Permanent address: Department of Computer Science
Old Dominion University
Norfolk, VA 23529
USA

This dissertation was typeset with \LaTeX^\dagger by the author.

[†] \LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's \TeX Program.